# JavaEE

## JavaEE的三层模型

### Domain Object(领域对象）层

此层由一系列的POJO（Plain Old Java Object，普通的、传统的Java对象）组成，这些对象是该系统的Domain Object（领域对象），往往包含了各自所需实现的业务逻辑方法。

### DAO（Data Access Object,数据访问对象）层

此层由一系列的DAO组件组成，这些DAO实现了对数据库的创建、查询、更新和删除（CRUD）等原子操作。

注：

1、在经典JavaEE应用中，DAO层也被称为EAO层，EAO层组件的作用与DAO层组件的作用基本相似。只是EAO层主要完成对实体（Entity)的CRUD操作，因此简称为EAO层。

2、DAO层在MyBatis中也被称为Mapper层，其通过SQL语句的映射完成CRUD操作。

### Service(业务逻辑）层

此层由一系列的业务逻辑对象组成，这些业务逻辑对象实现了系统所需要的业务逻辑方法。这些业务逻辑方法可能仅仅用于暴露 Domain Object对象所实现的业务逻辑方法，也可能是依赖DAO组件实现的业务逻辑方法。

### Controller（控制器）层

此层由一系列控制器组成，这些控制器用于拦截用户请求，并调用业务逻辑组件的业务逻辑方法，处理用户请求，并根据处理结果向不同的表现层组件转发。

 Controller，从字面上理解是控制器，所以它是负责业务调度的，所以在这一层应写一些业务的调度代码，而具体的业务处理应放在service中去写，而且service不单纯是对于dao的增删改查的调用，service是业务层，所以应该更切近于具体业务功能要求，所以在这一层，一个方法所体现的是一个可以对外提供的功能，比如购物商城中的生成订单方法，这里面就不简单是增加个订单记录那么简单，我们需要查询库存，核对商品等一系列实际业务逻辑的处理；

### View(表现）层

此层由一系列的JSP页面、Velocity页面、PDF文档视图组件组成负责收集用户请求，并显示处理结果。

## JavaEE的组件

### 表现层组件

主要负责收集用户输入数据，或者向客户显示系统状态。最常用的表现层技术是JSP，但JSP并不是唯一的表现层技术。表现层还可由Velocity、FreeMarker和Tapestry等技术完成，或者使用普通的应用程序充当表现层组件，甚至可以是小型智能设备。

### 控制器组件

对于JavaEE的MVC框架，其提供一个前端核心控制器，核心控制器负责拦截用户请求，并将请求转发给用户实现的控制器组件。这些用户实现的控制器组件则负责调用业务逻辑方法，处理用户请求。

### 业务逻辑组件

这是系统的核心组件，实现系统的业务逻辑。通常，一个业务逻辑方法对应一次用户操作。一个业务逻辑方法应该是一个整体，因此要求对业务逻辑方法增加事务性。业务逻辑方法仅仅负责实现业务逻辑，不应该进行数据库访问。因此，业务逻辑组件中不应该出现原始的MyBatis、Hibernate和JDBC等API。

# spring概述

## Spring概述

### spring框架概述

 Spring是一个开源框架，Spring是于2003 年兴起的一个轻量级的Java 开发框架，由Rod Johnson 在其著作Expert One-On-One J2EE Development and Design中阐述的部分理念和原型衍生而来。它是为了解决企业应用开发的复杂性而创建的。框架的主要优势之一就是其分层架构，分层架构允许使用者选择使用哪一个组件，同时为 J2EE 应用程序开发提供集成的框架。Spring使用基本的JavaBean来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。Spring的核心是控制反转（IoC）和面向切面（AOP）。简单来说，Spring是一个分层的JavaSE/EE full-stack(一站式) 轻量级开源框架。

轻量级：与EJB对比，依赖资源少，销毁的资源少。

分层： 一站式，每一个层都提供的解决方案

web层：struts，spring-MVC

service层：spring

dao层：hibernate，mybatis ， jdbcTemplate --> spring-data

### spring由来

 Expert One-to-One J2EE Design and Development

 Expert One-to-One J2EE Development without EJB

### spring核心

1，AOP（Aspect Oriented Programming）：面向切面编程。扩展功能而不修改源代码。

2，IOC：控制反转，降低耦合度。如果调用一个类中的方法，需要new对象然后才可以调用；在spring的IoC种，可以将代码new对象的操作交给spring的配置文件来完成。

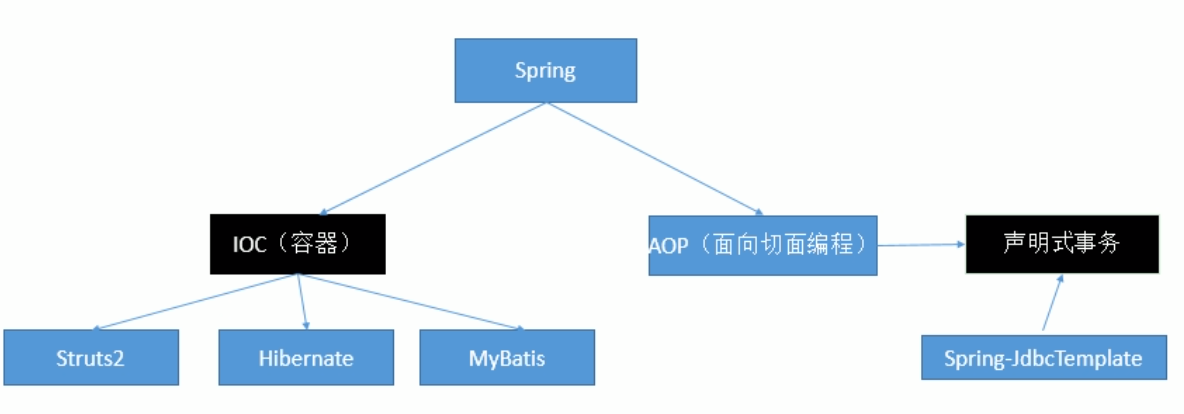
### spring优点



|  |
| --- |
| 方便解耦，简化开发 （高内聚低耦合）   * + Spring就是一个大工厂（容器），可以将所有对象创建和依赖关系维护，交给Spring管理   + spring工厂是用于生成bean   AOP编程的支持   * + Spring提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能   声明式事务的支持   * + 只需要通过配置就可以完成对事务的管理，而无需手动编程   方便程序的测试   * + Spring对Junit4支持，可以通过注解方便的测试Spring程序   方便集成各种优秀框架   * + Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如：Struts、Hibernate、MyBatis、Quartz等）的直接支持   降低JavaEE API的使用难度   * + Spring 对JavaEE开发中非常难用的一些API（JDBC、JavaMail、远程调用等），都提供了封装，使这些API应用难度大大降低 |

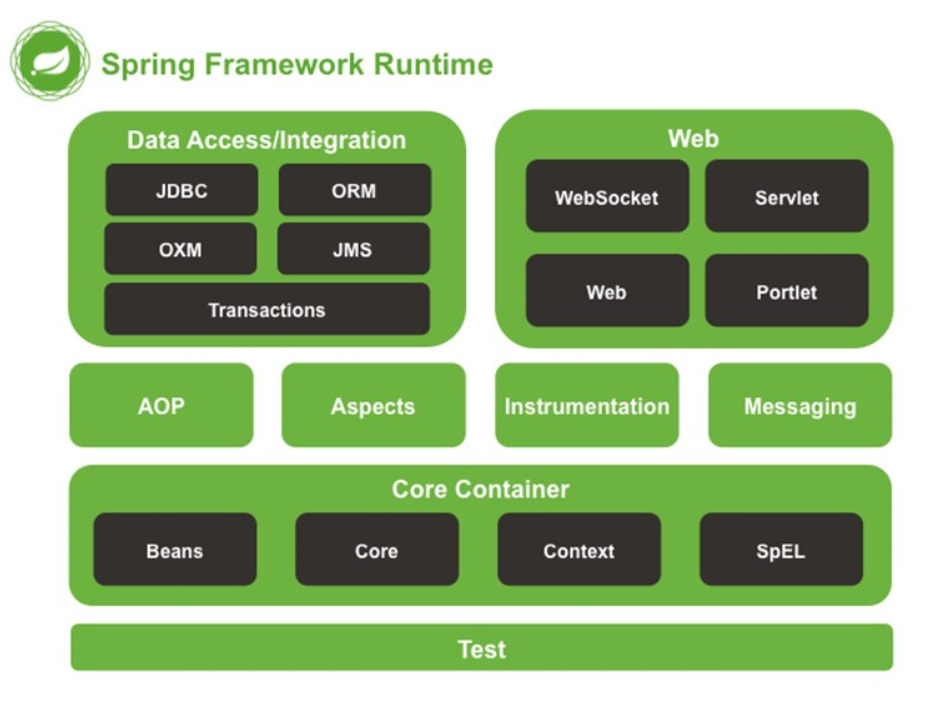
### 体系结构

核心容器：beans、core、context、expression



### 框架图

用那个模块导入那个包！



Test：spring的单元测试模块

spring-test-4.0.0.RELEASE

Core Container:核心容器（lOC）；黑色代表这部分的功能由哪些jar包组成；要使用这个部分的完整功能，这些jar都需要导入。

spring-beans-4.0.0.RELEASE

spring-core-4.0.0.RELEASE

spring-context-4.0.0.RELEASE

spring-expression-4.0.0.RELEASEI

AOP+Aspects(面向切面编程模块）

spring-aop-4.0.0.RELEASE

spring-aop-4.0.0.RELEASE

数据库访问模块

spring-jdbc-4.0.0.RELEASE、

spring-orm(Object Relation Mapping)-4.0.0.RELEASEH

spring的日志包

commons-loggin-1.1.3.jar

integration

spring-ox(xml)m-4.0.0.RELEASE

spring-jms-4.0.0.RELEASE、(Intergration)

web模块

spring-websocket(新的技术）-4.0.0.RELEASE、

spring-web-4.0.0.RELEASE、和原生的web相关（servlet）

spring-webmvc-4.0.0.RELEASE、开发web项目的（web)

spring-webmvc-portlet-4.0.0.RELEASE(开发web应用的组件集成）

## 相关概念说明



### 控制反转 （inversion of control）

谁控制谁? 控制什么? 为何叫反转(对应于正向)?哪些方面反转了?为何需要反转?

　　谁控制谁? --> IoC/DI容器控制应用程序

　　控制什么? --> IoC/DI容器控制对象本身的创建、实例化; IoC/DI容器控制对象之间的依赖关系

　　为何叫反转(对应于正向)? --> 因为现在应用程序不能主动去获取外部资源了，而是被动等待IoC/DI容器给它注入它所需要的资源，所以称之为反转.

　　哪些方面反转了? --> 1.创建对象 2.程序获取资源的方式反了

　　为何需要反转? --> 1.引入IoC/DI容器过后,体系更为松散，而且管理更有序; 2.类之间真正实现了松散耦合

传统的Java开发过程中，当需要一个对象的时候，我们需要自己new或者getInstance等直接或间接调用构造方法创建一个对象；而在Spring容器中，Spring容器利用 工厂模式(BeanFactory)为我们创建了所需要的对象，使用时无须自己创建，直接调用Spring为我们创建的对象即可，这就是控制反转的思想。

优点：简化了我们对对象的管理，交给了Spring管理；程序员不用再自己创建对象，需要时直接从Spring中取。由主动创建对象变成了被动接收对象，控制权反转，减轻了程序员的代码量，提高了开发效率。打个比方：IOC就像现在得订餐APP，以前我们吃饭需要自己去做，去New一顿饭出来；现在我们有了订餐APP(Spring)，我们只需要登录这个APP，在APP订餐页面applicationtext.xml(Spring配置文件)填写你想吃的饭(也就是Bean对象)，然后就可以预定你要吃的饭，想要猪头肉就会给你送猪头肉，想要红烧肉就会送红烧肉；各种菜就像是Property，这样我们坐在家中就可以吃到饭了，不用自己再动手去做了。

注意应用控制反转后，当对象被创建时，由一个调控系统内的所有对象的外界实体将其所依赖的对象的引用传递给它，即依赖被注入到对象中。所以，控制反转是关于一个对象如何获取它所依赖的对象的引用，在这里，反转指的是责任的反转。

### 依赖

什么是依赖(按名称理解、按动词理解)? 谁依赖于谁? 为什么需要依赖? 依赖什么东西?

　　什么是依赖(按名称理解、按动词理解)? --> 依赖(按名称理解):依赖关系; 依赖(按动词理解):依赖的动作

　　谁依赖于谁?　--> 应用程序依赖于IoC/DI容器

　　为什么需要依赖? --> 因为发生了反转，应用程序依赖的资源都是IoC/DI容器里面

　　依赖什么东西? --> 应用程序依赖于IoC/DI容器,依赖IoC/DI容器为它注入所需要的资源。（比如：依赖关系）

### 注入:

谁注入于谁? 注入什么东西? 为何要注入?

　　谁注入于谁? --> IoC/DI容器注入于应用程序

　　注入什么东西? --> 注入应用程序需要的外部资源，比如依赖关系

　　为何要注入? --> 因为程序要正常运行需要这些外部资源

### 依赖注入和控制反转是同一概念吗?

　　不是同一概念， 其实它们两个描述的是同一件事件，但是是从不同的角度来说:控制反转是从IoC/DI容器的角度；依赖注入是从应用程序的角度

　　控制反转的描述： IoC/DI容器反过来控制应用程序，控制应用程序锁所需要的外部资源（比如：外部资源）

　　依赖注入的描述： 应用程序依赖IoC/DI容器，依赖它注入所需要的外部资源。

### 参与者都有哪些?

　　IoC/DI容器、应用程序

### IoC/DI是什么?能做什么?怎么做?用在什么地方?

IoC/DI是什么?

IoC(Inversion of Control):就是使用IoC/DI容器反过来控制应用程序所需要的外部资源，是程序开发思想。

　　DI(Dependency Injection)：就是应用程序依赖IoC/DI容器来注入所需要的外部资源，也是程序的开发思想。

　　能做什么? --> 松散耦合对象

　　怎么做? --> 使用Spring框架，里面有实现好了的IoC/DI容器

　　用在什么地方? --> 凡是程序里面需要使用外部资源的情况，都可以考虑使用IoC/DI容器

### 什么是外部资源

对于一个类来讲，所谓的外部资源，就是指在自己类的内部不能得到或实现的东西，比如说:在类里面需要读取一个配置文件，那么这个配置文件就相当于这个类的外部资源。又比如：A类里面要调用B类，那么对于A类来讲B类就是外部资源。

### IoC容器

　　简单的理解就是：实现IoC思想，并提供对象创建、对象装配以及对象生命周期管理的软件就是IoC容器。

　对IoC的理解：

　　　　a. 应用程序无需主动new对象，而是描述对象应该如何被创建

　　　　b. 应用程序不需要主动装配对象之间的依赖关系,而是描述需要哪个服务，IoC容器会帮你装配，被动接受装配

　　　　c. 主动变被动，是一种让服务消费者不直接依赖于服务提供者的组件设计方式，是一种减少类与类之间依赖的设计原则

### 使用IoC/DI容器开发需要改变思路

　　a. 应用程序不主动创建对象，但是要描述创建它们的方式

　　b. 在应用程序代码中不直接进行服务的装配，但是要描述哪一个组件需要哪一项服务，由容器负责将这些装配在一起。也就是说：所有的组件都是被动的，组件初始化和专供都是由容器负责，应用程序只是在获取相应的组件后，实现应用的功能即可。

### IOC和DI使用的底层技术

Xml配置文件

dom4j解析xml

工厂设计模式

反射

注解

### 最终目地

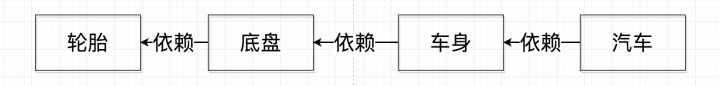
程序的高内聚，低耦合

## 控制反转和依赖注入详解

要了解**控制反转( Inversion of Control )**, 我觉得有必要先了解软件设计的一个重要思想：**依赖倒置原则（Dependency Inversion Principle）**。

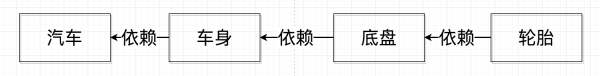
https://pic2.zhimg.com/v2-d53c75e91d959acbb0d95a835212ada5_b.png

**什么是依赖倒置原则？**假设我们设计一辆汽车：先设计轮子，然后根据轮子大小设计底盘，接着根据底盘设计车身，最后根据车身设计好整个汽车。这里就出现了一个“依赖”关系：汽车依赖车身，车身依赖底盘，底盘依赖轮子。



这样的设计看起来没问题，但是可维护性却很低。假设设计完工之后，上司却突然说根据市场需求的变动，要我们把车子的轮子设计都改大一码。这下我们就蛋疼了：因为我们是根据轮子的尺寸设计的底盘，轮子的尺寸一改，底盘的设计就得修改；同样因为我们是根据底盘设计的车身，那么车身也得改，同理汽车设计也得改——整个设计几乎都得改！

我们现在换一种思路。我们先设计汽车的大概样子，然后根据汽车的样子来设计车身，根据车身来设计底盘，最后根据底盘来设计轮子。这时候，依赖关系就倒置过来了：轮子依赖底盘， 底盘依赖车身， 车身依赖汽车。

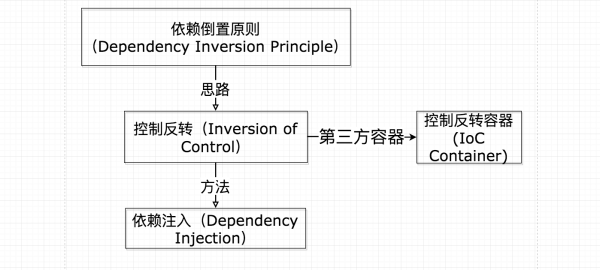


这时候，上司再说要改动轮子的设计，我们就只需要改动轮子的设计，而不需要动底盘，车身，汽车的设计了。

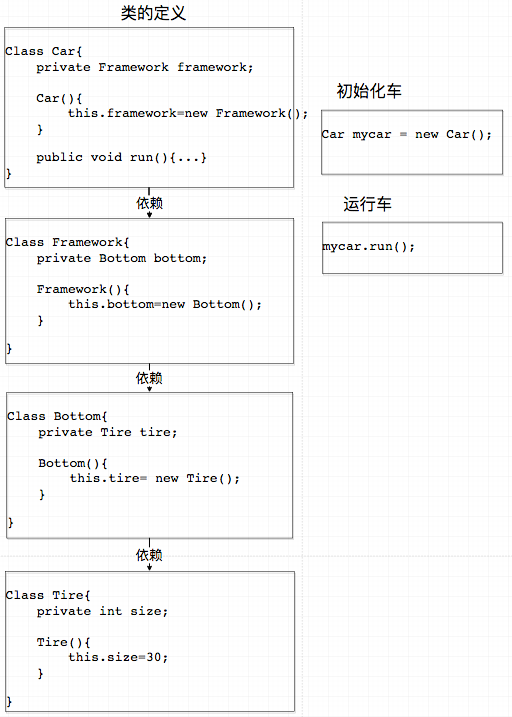
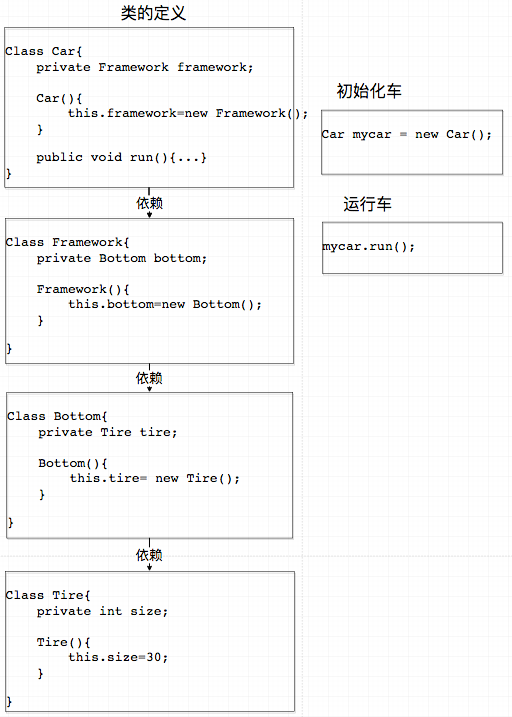
这就是依赖倒置原则——把原本的高层建筑依赖底层建筑“倒置”过来，变成底层建筑依赖高层建筑。高层建筑决定需要什么，底层去实现这样的需求，但是高层并不用管底层是怎么实现的。这样就不会出现前面的“牵一发动全身”的情况。

https://pic2.zhimg.com/v2-d53c75e91d959acbb0d95a835212ada5_b.png

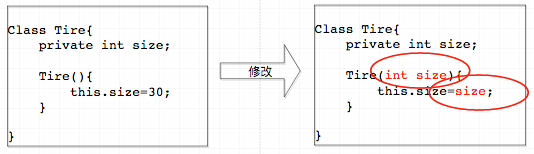
**控制反转（Inversion of Control）** 就是依赖倒置原则的一种代码设计的思路。具体采用的方法就是所谓的**依赖注入（Dependency Injection）**。其实这些概念初次接触都会感到云里雾里的。说穿了，这几种概念的关系大概如下：



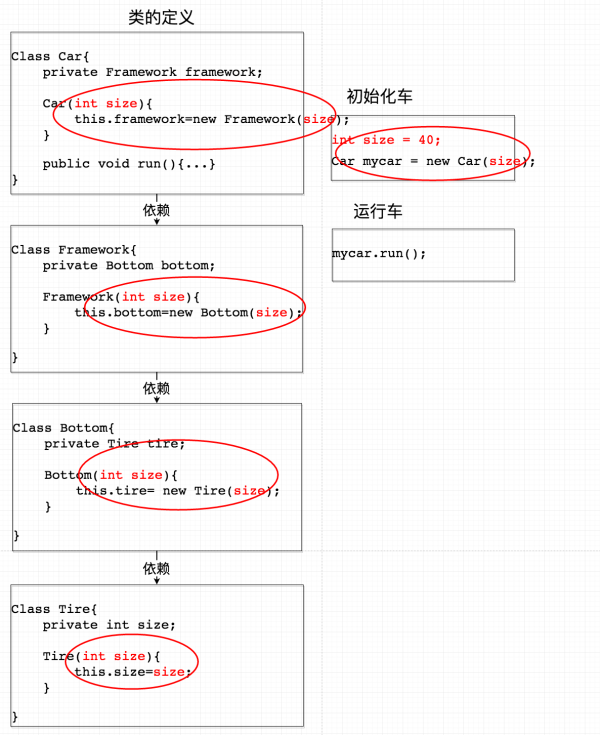
为了理解这几个概念，我们还是用上面汽车的例子。只不过这次换成代码。我们先定义四个Class，车，车身，底盘，轮胎。然后初始化这辆车，最后跑这辆车。代码结构如下：



这样，就相当于上面第一个例子，上层建筑依赖下层建筑——每一个类的构造函数都直接调用了底层代码的构造函数。假设我们需要改动一下轮胎（Tire）类，把它的尺寸变成动态的，而不是一直都是30。我们需要这样改：

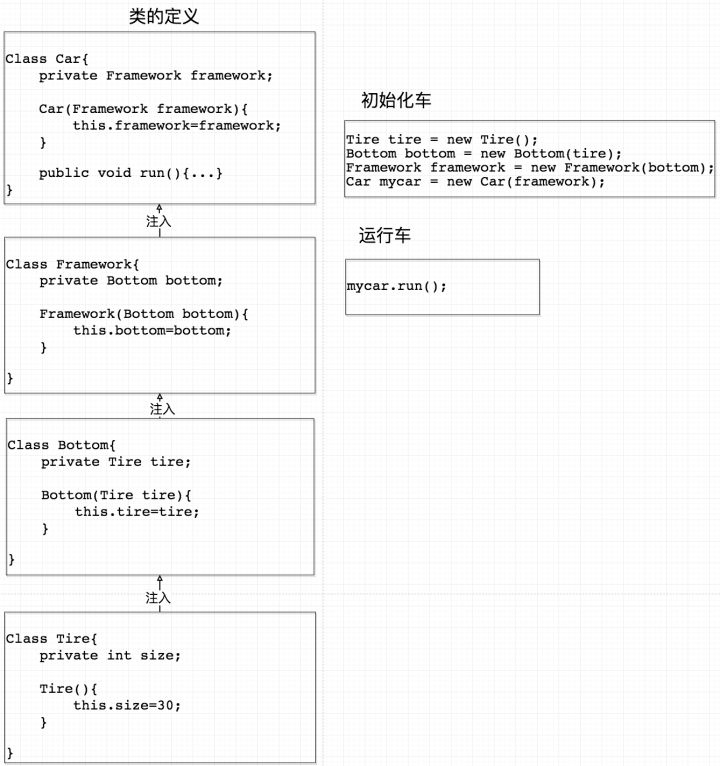


由于我们修改了轮胎的定义，为了让整个程序正常运行，我们需要做以下改动：



由此我们可以看到，仅仅是为了修改轮胎的构造函数，这种设计却需要**修改整个上层所有类的构造函数**！在软件工程中，**这样的设计几乎是不可维护的**——在实际工程项目中，有的类可能会是几千个类的底层，如果每次修改这个类，我们都要修改所有以它作为依赖的类，那软件的维护成本就太高了。

所以我们需要进行控制反转（IoC），及上层控制下层，而不是下层控制着上层。我们用依赖注入（Dependency Injection）这种方式来实现控制反转。**所谓依赖注入，就是把底层类作为参数传入上层类，实现上层类对下层类的“控制**”。这里我们用**构造方法传递的依赖注入方式**重新写车类的定义：

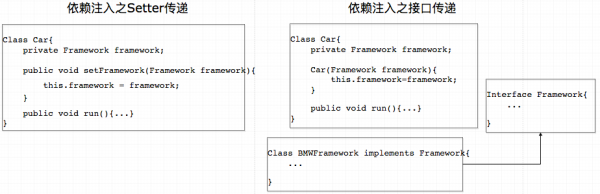


这里我们再把轮胎尺寸变成动态的，同样为了让整个系统顺利运行，我们需要做如下修改：



看到没？这里**我只需要修改轮胎类就行了，不用修改其他任何上层类。**这显然是更容易维护的代码。不仅如此，在实际的工程中，这种设计模式还有利于**不同组的协同合作和单元测试：**比如开发这四个类的分别是四个不同的组，那么只要定义好了接口，四个不同的组可以同时进行开发而不相互受限制；而对于单元测试，如果我们要写Car类的单元测试，就只需要Mock一下Framework类传入Car就行了，而不用把Framework, Bottom, Tire全部new一遍再来构造Car。

这里我们是采用的**构造函数传入**的方式进行的依赖注入。其实还有另外两种方法：**Setter传递**和**接口传递**。这里就不多讲了，核心思路都是一样的，都是为了实现**控制反转**。

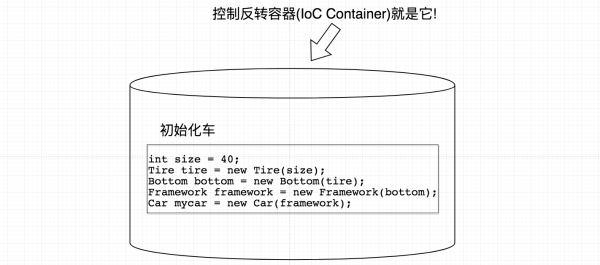


https://pic2.zhimg.com/v2-d53c75e91d959acbb0d95a835212ada5_b.png

# ioc容器

## 引入

看到这里你应该能理解什么控制反转和依赖注入了。那什么是**控制反转容器(IoC Container)**呢？其实上面的例子中，对车类进行初始化的那段代码发生的地方，就是控制反转容器。



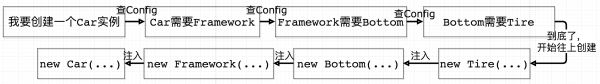
显然你也应该观察到了，因为采用了依赖注入，在初始化的过程中就不可避免的会写大量的new。这里IoC容器就解决了这个问题。**这个容器可以自动对你的代码进行初始化，你只需要维护一个Configuration（可以是xml可以是一段代码），而不用每次初始化一辆车都要亲手去写那一大段初始化的代码**。这是引入IoC Container的第一个好处。

IoC Container的第二个好处是：**我们在创建实例的时候不需要了解其中的细节。**在上面的例子中，我们自己手动创建一个车instance时候，是从底层往上层new的：

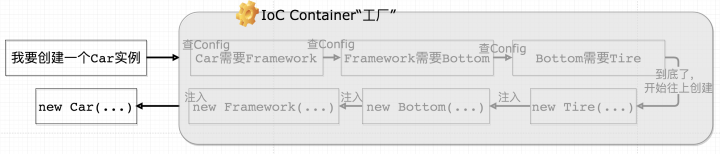
https://pic4.zhimg.com/v2-555b2be7d76e78511a6d6fed3304927f_b.png

这个过程中，我们需要了解整个Car/Framework/Bottom/Tire类构造函数是怎么定义的，才能一步一步new/注入。

而IoC Container在进行这个工作的时候是反过来的，它先从最上层开始往下找依赖关系，到达最底层之后再往上一步一步new（有点像深度优先遍历）：



这里IoC Container可以直接隐藏具体的创建实例的细节，在我们来看它就像一个工厂：



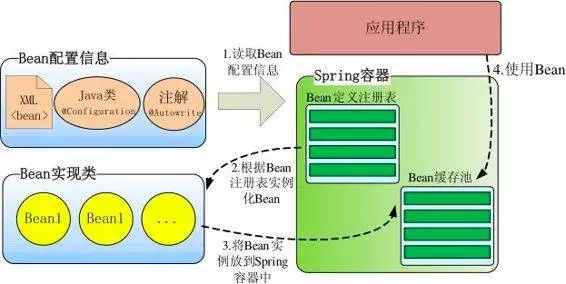
我们就像是工厂的客户。我们只需要向工厂请求一个Car实例，然后它就给我们按照Config创建了一个Car实例。我们完全不用管这个Car实例是怎么一步一步被创建出来。

实际项目中，有的Service Class可能是十年前写的，有几百个类作为它的底层。假设我们新写的一个API需要实例化这个Service，我们总不可能回头去搞清楚这几百个类的构造函数吧？IoC Container的这个特性就很完美的解决了这类问题——**因为这个架构要求你在写class的时候需要写相应的Config文件，所以你要初始化很久以前的Service类的时候，前人都已经写好了Config文件，你直接在需要用的地方注入这个Service就可以了**。这大大增加了项目的可维护性且降低了开发难度。

## IOC容器的初始化-过程

### Spring容器高层视图

Spring 启动时读取应用程序提供的Bean配置信息，并在Spring容器中生成一份相应的Bean配置注册表，然后根据这张注册表实例化Bean，装配好Bean之间的依赖关系，为上层应用提供准备就绪的运行环境。



Bean缓存池：HashMap实现

### IOC容器介绍

Spring 通过一个配置文件描述 Bean 及 Bean 之间的依赖关系，利用 Java 语言的反射功能实例化 Bean 并建立 Bean 之间的依赖关系。 Spring 的 IoC 容器在完成这些底层工作的基础上，还提供了 Bean 实例缓存、生命周期管理、 Bean 实例代理、事件发布、资源装载等高级服务。

BeanFactory 是 Spring 框架的基础设施，面向 Spring 本身；

ApplicationContext 面向使用 Spring 框架的开发者，几乎所有的应用场合我们都直接使用 ApplicationContext 而非底层的 BeanFactory。

ApplicationContext是BeanFactory的子接口；

BeanFactory:bean工厂接口；负责创建bean实例；容器里面保存的所有单例bean其实是一个map；Spring最底层的接口；

ApplicationContext:是容器接口；更多的负责容器功能的实现；（可以基于beanFactory创建好的对象之上完成强大的容器）容器可以从map获取这个bean，并且aop。di。在ApplicationContext接口的下的这些类里面；

BeanFactory最底层的接口，ApplicationContext留给程序员使用的ioc容器接口；ApplicationContext是BeanFactory的子接口；

## IOC源码分析

lOC：

1、IOC是一个容器

2、容器启动的时候创建所有单实例对象

3、我们可以直接从容器中获取到这个对象

SpringlOC:

1）、i2）、ioc是如何创建这些单实例bean，并如何管理的；到底保存在了那里？

oc容器的启动过程？启动期间都做了什么（什么时候创建所有单实例bean)

思路：

从HelloWorld开始，调试每个方法的作用；

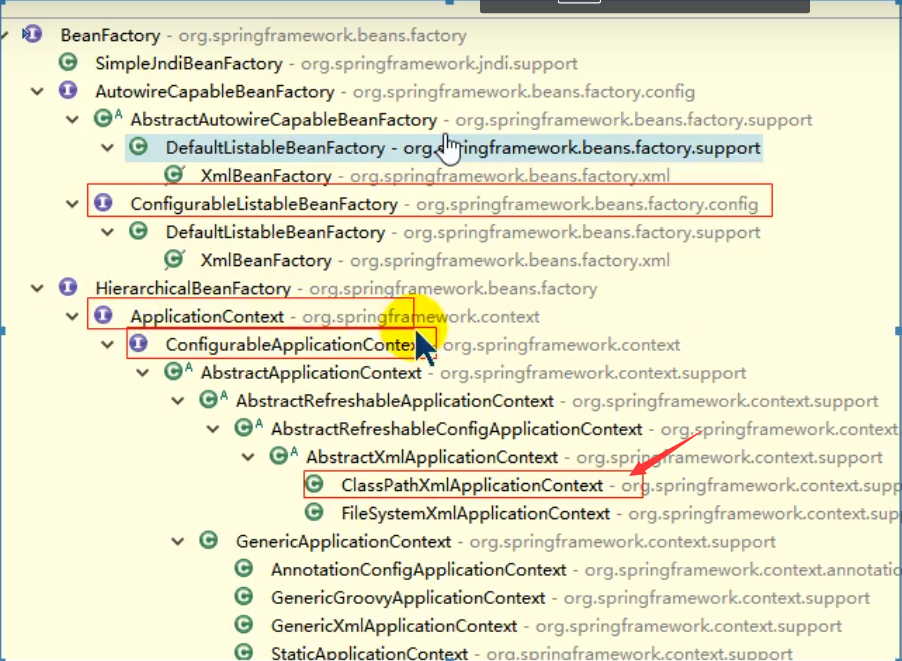
### ClassPathXMLApplicationContext构造器

|  |
| --- |
| ClassPathXmlApplicationContext *applicationContext* =  **new** ClassPathXmlApplicationContext("classpath:applicationContext.xml"); |

所有构造方法，都是调用下面这个方法。

|  |
| --- |
| **public** ClassPathXmlApplicationContext(  String[] configLocations, **boolean** refresh, @Nullable ApplicationContext parent)  **throws** BeansException {  **super**(parent);  setConfigLocations(configLocations); //设置配置文件的位置  **if** (refresh) {  refresh();//对象创建，所有单实例bena创建  }  } |

|  |
| --- |
| @Override  **public** **void** refresh() **throws** BeansException, IllegalStateException {  **synchronized** (**this**.startupShutdownMonitor) { //保证单实例  // Prepare this context for refreshing.  prepareRefresh();  //spring解析xml文件，并将信息保存起来  ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory(); //  prepareBeanFactory(beanFactory);  **try** {  postProcessBeanFactory(beanFactory);  invokeBeanFactoryPostProcessors(beanFactory);  //  registerBeanPostProcessors(beanFactory);  //支持国际化功能  initMessageSource();  initApplicationEventMulticaster();  //留给子类的  onRefresh();  //注册监听器  registerListeners();  //.初始化所有的的单实例  finishBeanFactoryInitialization(beanFactory);  finishRefresh();  }  **catch** (BeansException ex) {  **if** (logger.isWarnEnabled()) {  logger.warn("Exception encountered during context initialization - " +  "cancelling refresh attempt: " + ex);  }  // Destroy already created singletons to avoid dangling resources.  destroyBeans();  // Reset 'active' flag.  cancelRefresh(ex);  // Propagate exception to caller.  **throw** ex;  }  **finally** {  // Reset common introspection caches in Spring's core, since we  // might not ever need metadata for singleton beans anymore...  resetCommonCaches();  }  }  } |



finishBeanFactoryInitialization(beanFactory);

|  |
| --- |
| **protected** **void** finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {  **if** (beanFactory.containsBean(***CONVERSION\_SERVICE\_BEAN\_NAME***) &&  beanFactory.isTypeMatch(***CONVERSION\_SERVICE\_BEAN\_NAME***, ConversionService.**class**)) {  beanFactory.setConversionService(  beanFactory.getBean(***CONVERSION\_SERVICE\_BEAN\_NAME***, ConversionService.**class**));  }  **if** (!beanFactory.hasEmbeddedValueResolver()) {  beanFactory.addEmbeddedValueResolver(strVal -> getEnvironment().resolvePlaceholders(strVal));  }  String[] weaverAwareNames = beanFactory.getBeanNamesForType(LoadTimeWeaverAware.**class**, **false**, **false**);  **for** (String weaverAwareName : weaverAwareNames) {  getBean(weaverAwareName);  }  beanFactory.setTempClassLoader(**null**);  beanFactory.freezeConfiguration();  //初始化单实例bean  beanFactory. preInstantiateSingletons();  } |

preInstantiateSingletons();

|  |
| --- |
| @Override  **public** **void** preInstantiateSingletons() **throws** BeansException {  **if** (**this**.logger.isDebugEnabled()) {  **this**.logger.debug("Pre-instantiating singletons in " + **this**);  }  //拿到所有在xml中配置的bean的名字  List<String> beanNames = **new** ArrayList<String>(**this**.beanDefinitionNames);  //按顺序创建bean  **for** (String beanName : beanNames) {  //根据bean的名字获取bean的信息  RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);  //如果bean是单实例的，并且不是抽象的，不是懒加载  **if** (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) { //是工厂bean，用工工厂bean创建  **if** (isFactoryBean(beanName)) {  **final** FactoryBean<?> factory = (FactoryBean<?>) getBean(***FACTORY\_BEAN\_PREFIX*** + beanName);  **boolean** isEagerInit;  **if** (System.*getSecurityManager*() != **null** && factory **instanceof** SmartFactoryBean) {  isEagerInit = AccessController.*doPrivileged*(**new** PrivilegedAction<Boolean>() {  @Override  **public** Boolean run() {  **return** ((SmartFactoryBean<?>) factory).isEagerInit();  }  }, getAccessControlContext());  }  **else** {  isEagerInit = (factory **instanceof** SmartFactoryBean &&  ((SmartFactoryBean<?>) factory).isEagerInit());  }  **if** (isEagerInit) {  //创建bean  getBean(beanName);  }  }  **else** {  //创建bean  getBean(beanName);  }  }  }  // Trigger post-initialization callback for all applicable beans...  **for** (String beanName : beanNames) {  Object singletonInstance = getSingleton(beanName);  **if** (singletonInstance **instanceof** SmartInitializingSingleton) {  **final** SmartInitializingSingleton smartSingleton = (SmartInitializingSingleton) singletonInstance;  **if** (System.*getSecurityManager*() != **null**) {  AccessController.*doPrivileged*(**new** PrivilegedAction<Object>() {  @Override  **public** Object run() {  smartSingleton.afterSingletonsInstantiated();  **return** **null**;  }  }, getAccessControlContext());  }  **else** {  smartSingleton.afterSingletonsInstantiated();  }  }  }  } |

getBean(String name) ：所有的getBean都是在调用**return** doGetBean(name, **null**, **null**, **false**);

|  |
| --- |
| @Override  **public** Object getBean(String name) **throws** BeansException {  **return** doGetBean(name, **null**, **null**, **false**);  } |

doGetBean(

|  |
| --- |
| @SuppressWarnings("unchecked")  **protected** <T> T doGetBean(  **final** String name, **final** Class<T> requiredType, **final** Object[] args, **boolean** typeCheckOnly)  **throws** BeansException {  **final** String beanName = transformedBeanName(name);  Object bean;  // Eagerly check singleton cache for manually registered singletons.  //先从已经注册的所有单实例bean中有没有这个bean，第一次创建时没有的。  Object sharedInstance = getSingleton(beanName);  **if** (sharedInstance != **null** && args == **null**) {  **if** (logger.isDebugEnabled()) {  **if** (isSingletonCurrentlyInCreation(beanName)) {  logger.debug("Returning eagerly cached instance of singleton bean '" + beanName +  "' that is not fully initialized yet - a consequence of a circular reference");  }  **else** {  logger.debug("Returning cached instance of singleton bean '" + beanName + "'");  }  }  bean = getObjectForBeanInstance(sharedInstance, name, beanName, **null**);  }  **else** {  // Fail if we're already creating this bean instance:  // We're assumably within a circular reference.  **if** (isPrototypeCurrentlyInCreation(beanName)) {  **throw** **new** BeanCurrentlyInCreationException(beanName);  }  // Check if bean definition exists in this factory.  BeanFactory parentBeanFactory = getParentBeanFactory();  **if** (parentBeanFactory != **null** && !containsBeanDefinition(beanName)) {  // Not found -> check parent.  String nameToLookup = originalBeanName(name);  **if** (args != **null**) {  // Delegation to parent with explicit args.  **return** (T) parentBeanFactory.getBean(nameToLookup, args);  }  **else** {  // No args -> delegate to standard getBean method.  **return** parentBeanFactory.getBean(nameToLookup, requiredType);  }  }  **if** (!typeCheckOnly) {  markBeanAsCreated(beanName);  }  **try** {  **final** RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);  checkMergedBeanDefinition(mbd, beanName, args);  // Guarantee initialization of beans that the current bean depends on.  //拿到创建bean之前需要条件创建的bean，depends-on属性，如果有循环创建。  String[] dependsOn = mbd.getDependsOn();  **if** (dependsOn != **null**) {  **for** (String dependsOnBean : dependsOn) {  **if** (isDependent(beanName, dependsOnBean)) {  **throw** **new** BeanCreationException(mbd.getResourceDescription(), beanName,  "Circular depends-on relationship between '" + beanName + "' and '" + dependsOnBean + "'");  }  registerDependentBean(dependsOnBean, beanName);  getBean(dependsOnBean);  }  }  //创建bean实例  // Create bean instance.  **if** (mbd.isSingleton()) {  //  sharedInstance = getSingleton(beanName, **new** ObjectFactory<Object>() {  @Override  **public** Object getObject() **throws** BeansException {  **try** {  **return** createBean(beanName, mbd, args);  }  **catch** (BeansException ex) {  // Explicitly remove instance from singleton cache: It might have been put there  // eagerly by the creation process, to allow for circular reference resolution.  // Also remove any beans that received a temporary reference to the bean.  destroySingleton(beanName);  **throw** ex;  }  }  });  bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);  }  **else** **if** (mbd.isPrototype()) {  // It's a prototype -> create a new instance.  Object prototypeInstance = **null**;  **try** {  beforePrototypeCreation(beanName);  prototypeInstance = createBean(beanName, mbd, args);  }  **finally** {  afterPrototypeCreation(beanName);  }  bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);  }  **else** {  String scopeName = mbd.getScope();  **final** Scope scope = **this**.scopes.get(scopeName);  **if** (scope == **null**) {  **throw** **new** IllegalStateException("No Scope registered for scope name '" + scopeName + "'");  }  **try** {  Object scopedInstance = scope.get(beanName, **new** ObjectFactory<Object>() {  @Override  **public** Object getObject() **throws** BeansException {  beforePrototypeCreation(beanName);  **try** {  **return** createBean(beanName, mbd, args);  }  **finally** {  afterPrototypeCreation(beanName);  }  }  });  bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);  }  **catch** (IllegalStateException ex) {  **throw** **new** BeanCreationException(beanName,  "Scope '" + scopeName + "' is not active for the current thread; consider " +  "defining a scoped proxy for this bean if you intend to refer to it from a singleton",  ex);  }  }  }  **catch** (BeansException ex) {  cleanupAfterBeanCreationFailure(beanName);  **throw** ex;  }  }  // Check if required type matches the type of the actual bean instance.  **if** (requiredType != **null** && bean != **null** && !requiredType.isAssignableFrom(bean.getClass())) {  **try** {  **return** getTypeConverter().convertIfNecessary(bean, requiredType);  }  **catch** (TypeMismatchException ex) {  **if** (logger.isDebugEnabled()) {  logger.debug("Failed to convert bean '" + name + "' to required type [" +  ClassUtils.*getQualifiedName*(requiredType) + "]", ex);  }  **throw** **new** BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());  }  }  **return** (T) bean;  } |

DefaultSingletonBeanRegistry

//缓冲对象的map，也是ioc容器之一，保存单实例的。

/\*\* Cache of singleton objects: bean name --> bean instance \*/

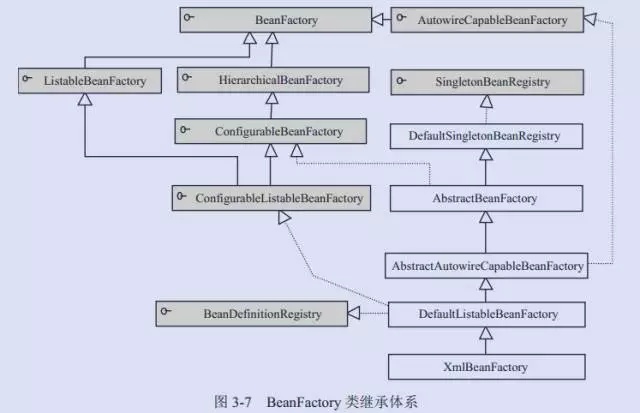
**private** **final** Map<String, Object> singletonObjects = **new** ConcurrentHashMap<String, Object>(256);

getSingleton(beanName, **new** ObjectFactory<Object>() {

|  |
| --- |
| **public** Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {  Assert.*notNull*(beanName, "'beanName' must not be null");  **synchronized** (**this**.singletonObjects) {  Object singletonObject = **this**.singletonObjects.get(beanName);  **if** (singletonObject == **null**) {  **if** (**this**.singletonsCurrentlyInDestruction) {  **throw** **new** BeanCreationNotAllowedException(beanName,  "Singleton bean creation not allowed while the singletons of this factory are in destruction " +  "(Do not request a bean from a BeanFactory in a destroy method implementation!)");  }  **if** (logger.isDebugEnabled()) {  logger.debug("Creating shared instance of singleton bean '" + beanName + "'");  }  beforeSingletonCreation(beanName);  **boolean** newSingleton = **false**;  **boolean** recordSuppressedExceptions = (**this**.suppressedExceptions == **null**);  **if** (recordSuppressedExceptions) {  **this**.suppressedExceptions = **new** LinkedHashSet<Exception>();  }  **try** {  singletonObject = singletonFactory.getObject();  newSingleton = **true**;  }  **catch** (IllegalStateException ex) {  // Has the singleton object implicitly appeared in the meantime ->  // if yes, proceed with it since the exception indicates that state. //利用反射创建对象  singletonObject = **this**.singletonObjects.get(beanName);  **if** (singletonObject == **null**) {  **throw** ex;  }  }  **catch** (BeanCreationException ex) {  **if** (recordSuppressedExceptions) {  **for** (Exception suppressedException : **this**.suppressedExceptions) {  ex.addRelatedCause(suppressedException);  }  }  **throw** ex;  }  **finally** {  **if** (recordSuppressedExceptions) {  **this**.suppressedExceptions = **null**;  }  afterSingletonCreation(beanName);  }  **if** (newSingleton) {  //将对象添加到map中  addSingleton(beanName, singletonObject);  }  }  **return** (singletonObject != ***NULL\_OBJECT*** ? singletonObject : **null**);  }  } |

## BeanFactory

BeanFactory体系架构：



BeanDefinitionRegistry： Spring 配置文件中每一个<bean>节点元素在 Spring 容器里都通过一个 BeanDefinition 对象表示，它描述了 Bean 的配置信息。而 BeanDefinitionRegistry 接口提供了向容器手工注册 BeanDefinition 对象的方法。

BeanFactory 接口位于类结构树的顶端 ，它最主要的方法就是 getBean(String beanName)，该方法从容器中返回特定名称的 Bean，BeanFactory 的功能通过其他的接口得到不断扩展：

ListableBeanFactory：该接口定义了访问容器中 Bean 基本信息的若干方法，如查看Bean 的个数、获取某一类型 Bean 的配置名、查看容器中是否包括某一 Bean 等方法；

HierarchicalBeanFactory：父子级联 IoC 容器的接口，子容器可以通过接口方法访问父容器； 通过 HierarchicalBeanFactory 接口， Spring 的 IoC 容器可以建立父子层级关联的容器体系，子容器可以访问父容器中的 Bean，但父容器不能访问子容器的 Bean。Spring 使用父子容器实现了很多功能，比如在 Spring MVC 中，展现层 Bean 位于一个子容器中，而业务层和持久层的 Bean 位于父容器中。这样，展现层 Bean 就可以引用业务层和持久层的 Bean，而业务层和持久层的 Bean 则看不到展现层的 Bean。

ConfigurableBeanFactory：是一个重要的接口，增强了 IoC 容器的可定制性，它定义了设置类装载器、属性编辑器、容器初始化后置处理器等方法；

AutowireCapableBeanFactory：定义了将容器中的 Bean 按某种规则（如按名字匹配、按类型匹配等）进行自动装配的方法；

SingletonBeanRegistry：定义了允许在运行期间向容器注册单实例 Bean 的方法；

例子：使用 Spring 配置文件为 Car 提供配置信息：beans.xml：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?>  <beans xmlns="Index of /schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:p="http://www.springframework.org/schema/p"  xsi:schemaLocation="Index of /schema/beans  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">    <bean id="car1" class="com.baobaotao.Car"  p:brand="红旗CA72"  p:color="黑色"  p:maxSpeed="200" />  </beans> |

通过 BeanFactory 装载配置文件，启动 Spring IoC 容器：

|  |
| --- |
| public class BeanFactoryTest {  public static void main(String[] args) throws Throwable{  ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();  Resource res = resolver.getResource("classpath:com/baobaotao/beanfactory/beans.xml");  BeanFactory bf = new XmlBeanFactory(res);  System.out.println("init BeanFactory.");  Car car = bf.getBean("car",Car.class);  System.out.println("car bean is ready for use!");  } |

XmlBeanFactory 通过 Resource 装载 Spring 配置信息并启动 IoC 容器，然后就可以通过 BeanFactory#getBean(beanName)方法从 IoC 容器中获取 Bean 了。通过 BeanFactory 启动IoC 容器时，并不会初始化配置文件中定义的 Bean，初始化动作发生在第一个调用时。

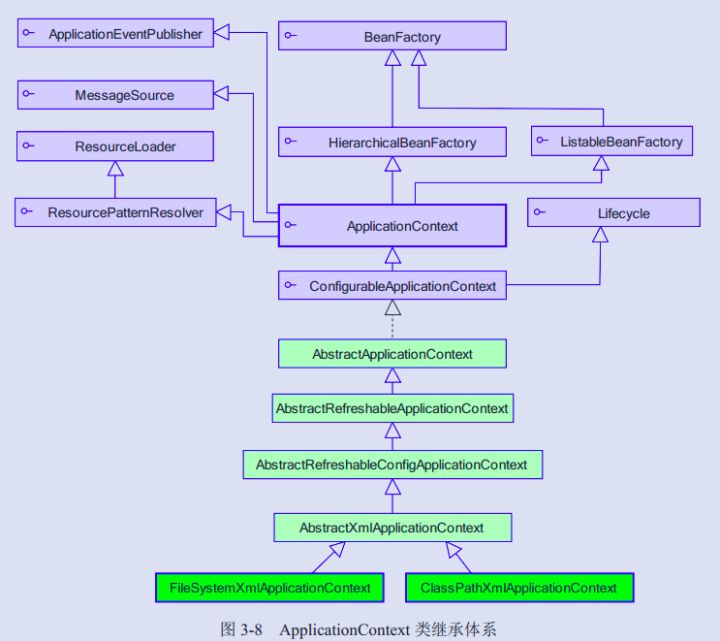
对于单实例（ singleton）的 Bean 来说，BeanFactory会缓存 Bean 实例，所以第二次使用 getBean() 获取 Bean 时将直接从 IoC 容器的缓存中获取 Bean 实例。Spring 在 DefaultSingletonBeanRegistry 类中提供了一个用于缓存单实例 Bean 的缓存器，它是一个用HashMap 实现的缓存器，单实例的 Bean 以 beanName 为键保存在这个HashMap 中。

值得一提的是，在初始化 BeanFactory 时，必须为其提供一种日志框架，比如使用Log4J， 即在类路径下提供 Log4J 配置文件，这样启动 Spring 容器才不会报错。

## ApplicationContext

ApplicationContext 由 BeanFactory 派生而来，提供了更多面向实际应用的功能。

在BeanFactory 中，很多功能需要以编程的方式实现，而在 ApplicationContext 中则可以通过配置的方式实现。



ClassPathXmlApplicationContext：默认从类路径加载配置文件

FileSystemXmlApplicationContext：默认从文件系统中装载配置文件

ApplicationEventPublisher：让容器拥有发布应用上下文事件的功能，包括容器启动事件、关闭事件等。实现了 ApplicationListener 事件监听接口的 Bean 可以接收到容器事件 ， 并对事件进行响应处理 。 在 ApplicationContext 抽象实现类AbstractApplicationContext 中，我们可以发现存在一个 ApplicationEventMulticaster，它负责保存所有监听器，以便在容器产生上下文事件时通知这些事件监听者。

MessageSource：为应用提供 i18n 国际化消息访问的功能；

ResourcePatternResolver ： 所 有 ApplicationContext 实现类都实现了类似于PathMatchingResourcePatternResolver 的功能，可以通过带前缀的 Ant 风格的资源文件路径装载 Spring 的配置文件。

LifeCycle：该接口是 Spring 2.0 加入的，该接口提供了 start()和 stop()两个方法，主要用于控制异步处理过程。在具体使用时，该接口同时被 ApplicationContext 实现及具体 Bean 实现， ApplicationContext 会将 start/stop 的信息传递给容器中所有实现了该接口的 Bean，以达到管理和控制 JMX、任务调度等目的。

ConfigurableApplicationContext 扩展于 ApplicationContext，它新增加了两个主要的方法： refresh()和 close()，让 ApplicationContext 具有启动、刷新和关闭应用上下文的能力。在应用上下文关闭的情况下调用 refresh()即可启动应用上下文，在已经启动的状态下，调用 refresh()则清除缓存并重新装载配置信息，而调用close()则可关闭应用上下文。这些接口方法为容器的控制管理带来了便利，但作为开发者，我们并不需要过多关心这些方法。

使用：

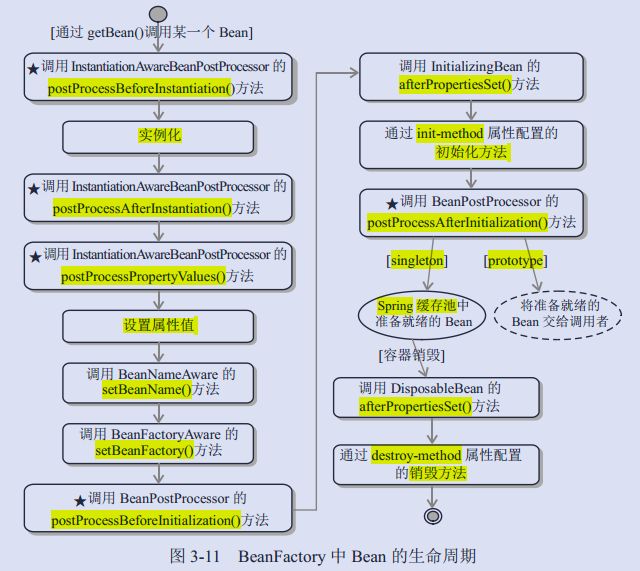
如果配置文件放置在类路径下，用户可以优先使用 ClassPathXmlApplicationContext 实现类：

|  |
| --- |
| ApplicationContext ctx = new ClassPathXmlApplicationContext("com/baobaotao/ context/beans.xml"); |

如果配置文件放置在文件系统的路径下，则可以优先考虑使用 FileSystemXmlApplicationContext 实现类：

|  |
| --- |
| ApplicationContext ctx = new FileSystemXmlApplicationContext("com/baobaotao/ context/beans.xml"); |

## Bean的生命周期



1．当调用者通过 getBean(beanName)向容器请求某一个 Bean 时，如果容器注册了org.springframework.beans.factory.config.InstantiationAwareBeanPostProcessor 接口，在实例化 Bean 之前，将调用接口的 postProcessBeforeInstantiation()方法；

2．根据配置情况调用 Bean 构造函数或工厂方法实例化 Bean；

3．如果容器注册了 InstantiationAwareBeanPostProcessor 接口，在实例化 Bean 之后，调用该接口的 postProcessAfterInstantiation()方法，可在这里对已经实例化的对象进行一些“梳妆打扮”；

4．如果 Bean 配置了属性信息，容器在这一步着手将配置值设置到 Bean 对应的属性中，不过在设置每个属性之前将先调用InstantiationAwareBeanPostProcessor 接口的postProcessPropertyValues()方法；

5．调用 Bean 的属性设置方法设置属性值；

6．如果 Bean 实现了 org.springframework.beans.factory.BeanNameAware 接口，将调用setBeanName()接口方法，将配置文件中该 Bean 对应的名称设置到 Bean 中；

7．如果 Bean 实现了 org.springframework.beans.factory.BeanFactoryAware 接口，将调用 setBeanFactory()接口方法，将 BeanFactory 容器实例设置到 Bean 中；

8．如果 BeanFactory 装配了 org.springframework.beans.factory.config.BeanPostProcessor后处理器，将调用 BeanPostProcessor 的 Object postProcessBeforeInitialization(Object bean, String beanName)接口方法对 Bean 进行加工操作。其中入参 bean 是当前正在处理的 Bean，而 beanName 是当前 Bean 的配置名，返回的对象为加工处理后的 Bean。用户可以使用该方法对某些 Bean 进行特殊的处理，甚至改变 Bean 的行为， BeanPostProcessor 在 Spring 框架中占有重要的地位，为容器提供对 Bean 进行后续加工处理的切入点， Spring 容器所提供的各种“神奇功能”（如 AOP，动态代理等）都通过 BeanPostProcessor 实施；

9．如果 Bean 实现了 InitializingBean 的接口，将调用接口的 afterPropertiesSet()方法；

10．如果在<bean>通过 init-method 属性定义了初始化方法，将执行这个方法；

11．BeanPostProcessor 后处理器定义了两个方法：其一是 postProcessBeforeInitialization() 在第 8 步调用；其二是 Object postProcessAfterInitialization(Object bean, String beanName)方法，这个方法在此时调用，容器再次获得对 Bean 进行加工处理的机会；

12．如果在<bean>中指定 Bean 的作用范围为 scope=“prototype”，将 Bean 返回给调用者，调用者负责 Bean 后续生命的管理， Spring 不再管理这个 Bean 的生命周期。如果作用范围设置为 scope=“singleton”，则将 Bean 放入到 Spring IoC 容器的缓存池中，并将 Bean引用返回给调用者， Spring 继续对这些 Bean 进行后续的生命管理；

13．对于 scope=“singleton”的 Bean，当容器关闭时，将触发 Spring 对 Bean 的后续生命周期的管理工作，首先如果 Bean 实现了 DisposableBean 接口，则将调用接口的afterPropertiesSet()方法，可以在此编写释放资源、记录日志等操作；

14．对于 scope=“singleton”的 Bean，如果通过<bean>的 destroy-method 属性指定了 Bean 的销毁方法， Spring 将执行 Bean 的这个方法，完成 Bean 资源的释放等操作。

可以将这些方法大致划分为三类：

Bean 自身的方法：如调用 Bean 构造函数实例化 Bean，调用 Setter 设置 Bean 的属性值以及通过<bean>的 init-method 和 destroy-method 所指定的方法；

Bean 级生命周期接口方法：如 BeanNameAware、 BeanFactoryAware、 InitializingBean 和 DisposableBean，这些接口方法由 Bean 类直接实现；

容器级生命周期接口方法：在上图中带“★” 的步骤是由 InstantiationAwareBean PostProcessor 和 BeanPostProcessor 这两个接口实现，一般称它们的实现类为“ 后处理器” 。 后处理器接口一般不由 Bean 本身实现，它们独立于 Bean，实现类以容器附加装置的形式注册到 Spring 容器中并通过接口反射为 Spring 容器预先识别。当Spring 容器创建任何 Bean 的时候，这些后处理器都会发生作用，所以这些后处理器的影响是全局性的。当然，用户可以通过合理地编写后处理器，让其仅对感兴趣Bean 进行加工处理

## 容器启动过程

web环境下Spring容器、SpringMVC容器启动过程：

首先，对于一个web应用，其部署在web容器中，web容器提供其一个全局的上下文环境，这个上下文就是ServletContext，其为后面的spring IoC容器提供宿主环境；

其次，在web.xml中会提供有contextLoaderListener（或ContextLoaderServlet）。在web容器启动时，会触发容器初始化事件，此时contextLoaderListener会监听到这个事件，其contextInitialized方法会被调用，在这个方法中，spring会初始化一个启动上下文，这个上下文被称为根上下文，即WebApplicationContext，这是一个接口类，确切的说，其实际的实现类是XmlWebApplicationContext。这个就是spring的IoC容器，其对应的Bean定义的配置由web.xml中的context-param标签指定。在这个IoC容器初始化完毕后，spring容器以WebApplicationContext.ROOTWEBAPPLICATIONCONTEXTATTRIBUTE为属性Key，将其存储到ServletContext中，便于获取；

再次，contextLoaderListener监听器初始化完毕后，开始初始化web.xml中配置的Servlet，这个servlet可以配置多个，以最常见的DispatcherServlet为例（Spring MVC），这个servlet实际上是一个标准的前端控制器，用以转发、匹配、处理每个servlet请求。DispatcherServlet上下文在初始化的时候会建立自己的IoC上下文容器，用以持有spring mvc相关的bean，这个servlet自己持有的上下文默认实现类也是XmlWebApplicationContext。在建立DispatcherServlet自己的IoC上下文时，会利用WebApplicationContext.ROOTWEBAPPLICATIONCONTEXTATTRIBUTE先从ServletContext中获取之前的根上下文(即WebApplicationContext)作为自己上下文的parent上下文（即第2步中初始化的XmlWebApplicationContext作为自己的父容器）。有了这个parent上下文之后，再初始化自己持有的上下文（这个DispatcherServlet初始化自己上下文的工作在其initStrategies方法中可以看到，大概的工作就是初始化处理器映射、视图解析等）。初始化完毕后，spring以与servlet的名字相关(此处不是简单的以servlet名为Key，而是通过一些转换)的属性为属性Key，也将其存到ServletContext中，以便后续使用。这样每个servlet就持有自己的上下文，即拥有自己独立的bean空间，同时各个servlet共享相同的bean，即根上下文定义的那些bean。

## Bean加载过程

Spring的高明之处在于，它使用众多接口描绘出了所有装置的蓝图，构建好Spring的骨架，继而通过继承体系层层推演，不断丰富，最终让Spring成为有血有肉的完整的框架。所以查看Spring框架的源码时，有两条清晰可见的脉络：

1）接口层描述了容器的重要组件及组件间的协作关系；

2）继承体系逐步实现组件的各项功能。

接口层清晰地勾勒出Spring框架的高层功能，框架脉络呼之欲出。有了接口层抽象的描述后，不但Spring自己可以提供具体的实现，任何第三方组织也可以提供不同实现， 可以说Spring完善的接口层使框架的扩展性得到了很好的保证。纵向继承体系的逐步扩展，分步骤地实现框架的功能，这种实现方案保证了框架功能不会堆积在某些类的身上，造成过重的代码逻辑负载，框架的复杂度被完美地分解开了。

# 入门程序

## 导包

### 核心包



### 日志包

导入jar包



配置log4j.properties文件//不用修改

|  |
| --- |
| # Global logging configuration  log4j.rootLogger=DEBUG, stdout  # MyBatis logging configuration...  log4j.logger.cn.sxt.mapper=TRACE  # Console output...  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n |

## 入门案例IOC(控制反转)

目标：提供一个dao接口、或的该接口的实现类对象。

效果：将由Spring创建对象实例--> IoC 控制反转（Inverse of Control）之后需要实例对象时，从spring工厂(容器)中获得，需要将实现类的全限定名称配置到xml文件中。

### 接口

|  |
| --- |
| //获取用户名  **public** **void** getUserName(); |

### 实现接口的类

|  |
| --- |
| **public** **class** UserDaoImpl **implements** UserDao {  @Override  **public** **void** getUserName() {  System.*out*.println("通过spring创建对象调用");  }  } |

### applicationContext.xml配置文件

位置：config文件下

名称：任意，开发中常用applicationContext.xml

内容：添加schema约束

约束文件位置：spring-framework-3.2.0.RELEASE\docs\spring-framework-reference\html\ xsd-config.html

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  约束  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd"*>  <!-- 配置daoimple  <bean> 配置需要创建的对象  id ：用于之后从spring容器获得实例时使用的  class ：需要创建实例的全限定类名  -->  <bean id=*"daoimpl"* class=*"com.sx.daoImpl.UserDaoImpl"*>  </bean>  </beans> |

### 测试

|  |
| --- |
| **public** **void** test1(){  //1、加载配置文件，并获得容器。  ApplicationContext applicationContext =**new** ClassPathXmlApplicationContext("classpath:applicationContext.xml");  //2获得内容 --不需要自己new，通过配置文件中bean的id获取创建的userdaoimpl对象。  UserDaoImpl udi =(UserDaoImpl)applicationContext.getBean("daoimpl");  udi.getUserName();  } |

## 入门案例DI（Dependency Injection）

依赖注入（Dependency Injection，简称DI）：在程序运行过程中，客户类不直接实例化具体服务类实例，而是客户类的运行上下文环境或专门组件负责实例化服务类，然后将其注入到客户类中，保证客户类的正常运行。

在这里指的是 要创建的对象中的成员变量由 Spring 进行 set。

### applicationContext.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  <beanid="UserService"class="com.ittianyu.spring.a\_base.z\_common.service.impl.UserServiceImpl">  <!-- 依赖注入 DI：在创建 service 时，需要设置 service 所依赖的 dao 实现 -->  <property name="userDao" ref="UserDao"/>  </bean>  <bean id="UserDao" class="com.ittianyu.spring.a\_base.z\_common.dao.impl.UserDaoImpl"/>  </beans> |

### test

|  |
| --- |
| public class TestClass {  @Test  public void helloDI() {  String xmlPath = "com/ittianyu/spring/b\_di/applicationContext.xml";  ApplicationContext applicationContext = new ClassPathXmlApplicationContext(xmlPath);  // 通过 spring 创建 service  UserService userService = (UserService) applicationContext.getBean("UserService");  userService.addUser(new User("hello spring !"));  }  } |

# spring的相关配置

## Bean标签

### bean的种类

普通bean：之前操作的都是普通bean。<bean id="" class="A"> ，spring直接创建A实例，并返回

FactoryBean：是一个特殊的bean，具有工厂生成对象能力，只能生成特定的对象。

bean必须使用 FactoryBean接口，此接口提供方法 getObject() 用于获得特定bean。

<bean id="" class="FB">

先创建FB实例，使用调用getObject()方法，并返回方法的返回值

FB fb = new FB();

return fb.getObject();

BeanFactory 和 FactoryBean 对比？

BeanFactory：工厂，用于生成任意bean。

FactoryBean：特殊bean，用于生成另一个特定的bean。例如：ProxyFactoryBean ，此工厂bean用于生产代理。<bean id="" class="....ProxyFactoryBean"> 获得代理对象实例。AOP使用

### bean标签

1，bean标签id属性：为Bean标签自定义名称，但是不能使用特殊符号。在代码中可以根据id值获取到对象

2，bean标签class属性：代表对应类的全路径

3，bean标签scope属性：

    singleton：默认值，表示单例，创建对象的地址相同。

    prototype：多例

前后条交互的时候才是用下面

    request：WEB项目中，Spring创建一个Bean对象，将Bean对象放入request域中。

    session：WEB项目中，Spring创建一个Bean对象，将Bean对象放入session域中。

4，bean标签lazy属性

       默认情况下启动程序时对象就直接创建了，不管有没有使用bean对象都会调用了类无参的构造方法。

       当加入lazy-init=“true”之后，只有使用者取对象时才会创建对象，不会调用类无参的构造方法。

### autowire属性

当我们要往一个bean的某个属性里注入另外一个bean，我们会使用<property> + <ref/>标签的形式。但是对于大型项目，假设有一个bean A被多个bean引用注入，如果A的id因为某种原因修改了，那么所有引用了A的bean的<ref/>标签内容都得修改，这时候如果使用autowire="byType"，那么引用了A的bean就完全不用修改了。autowire到底都什么用？

用法：<bean id="auto" class="example.autoBean" autowire="byType"/>

五种模式

1、no

默认)不采用autowire机制.。这种情况，当我们需要使用依赖注入，只能用<ref/>标签。

2、byName

通过属性的名称自动装配（注入）。Spring会在容器中查找名称与bean属性名称一致的bean，并自动注入到bean属性中。当然bean的属性需要有setter方法。例如：bean A有个属性master，master的setter方法就是setMaster，A设置了autowire="byName"，那么Spring就会在容器中查找名为master的bean通过setMaster方法注入到A中。

3、Type

过类型自动装配（注入）。Spring会在容器中查找类（Class）与bean属性类一致的bean，并自动注入到bean属性中，如果容器中包含多个这个类型的bean，Spring将抛出异常。如果没有找到这个类型的bean，那么注入动作将不会执行。

4、constructor

类似于byType，但是是通过构造函数的参数类型来匹配。假设bean A有构造函数A(B b, C c)，那么Spring会在容器中查找类型为B和C的bean通过构造函数A(B b, C c)注入到A中。与byType一样，如果存在多个bean类型为B或者C，则会抛出异常。但时与byType不同的是，如果在容器中找不到匹配的类的bean，将抛出异常，因为Spring无法调用构造函数实例化这个bean。

5、default

采用父级标签（即beans的default-autowire属性）的配置

6、towire-candidate

前面我们说到配置有autowire属性的bean，Spring在实例化这个bean的时候会在容器中查找匹配的bean对autowire bean进行属性注入，这些被查找的bean我们称为候选bean。作为候选bean，我凭什么就要被你用，老子不给你用。所以候选bean给自己增加了autowire-candidate="false"属性（默认是true），那么容器就不会把这个bean当做候选bean了，即这个bean不会被当做自动装配对象。同样，<beans/>标签可以定义default-autowire-candidate="false"属性让它包含的所有bean都不做为候选bean。我的地盘我做主。

注：

其中byType和constructor模式也支持数组和强类型集合（即指定了集合元素类型）。如bean A有个属性定义是List<Foo>类型，Spring会在容器中查找所有类型为Foo的bean，注入到该属性。记住是Foo，不是List。

另外如果集合和Map集合，那么Map的key必须是String类型，Spring会根据value的类型去匹配。例如有属性bean A中有一个属性为Map<String, Foo> p，容器中有bean B和C类型均为Foo，那么A实例化完成后，p属性的值为：{"B"：B的实例对象，"C"：C的实例对象}。

除了可以给单独一个bean配置autowire属性，我们还可以给某个bean配置文件的<beans/>标签添加default-autowire属性。在<beans/>标签中指定default-autowire属性，那么对于子标签<bean/>如果没有单独的设置autowire属性，那么将采用父标签<beans/>的default-autowire属性的模式，如果单独设置了autowire属性，则采用自己的模式。

### parent属性

作用：bean配置的继承

|  |
| --- |
| <bean id=*"person1"* class=*"com.sx.domain.User"*>  <property name=*"name"* value=*"sx"*></property>  <property name=*"age"* value=*"12"*></property>  <property name=*"gender"* value=*"女"*></property>  </bean>  <bean id=*"person2"* class=*"com.sx.domain.User"* parent=*"person1"*>  <property name=*"gender"* value=*"男"*></property>  </bean> |

person2 从person1中继承自身中没有赋值的属性。

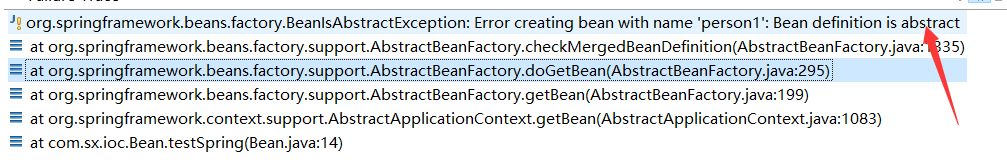
person2：User [name=sx, age=12, gender=男]

### abstract属性

abstract=true 表名这个bean是抽象的，不能获取到他的实例，只能被用来继承。

|  |
| --- |
| <bean id=*"person1"* class=*"com.sx.domain.User"* abstract=*"true"*>  <property name=*"name"* value=*"sx"*></property>  <property name=*"age"* value=*"12"*></property>  <property name=*"gender"* value=*"女"*></property>  </bean>  <bean id=*"person2"* class=*"com.sx.domain.User"* parent=*"person1"*>  <property name=*"gender"* value=*"男"*></property>  </bean> |

获取*person1*报错



### 生命周期方法

生命周期：bean的创建到销毁；

ioc容器中注册的bean；

1、单例bean，容器启动的时候就会创建好，容器关闭也会销毁创建的bean

2、多实例bean,获取的时候才创建，容器关闭不会调用bean的销毁方法

我们可以为bean自定义一些生命周期方法；spring在创建或者销毁的时候就会调用指定的方法；

注：初始化方法和销毁方法能够有参数。

|  |
| --- |
| <bean id=*"book"* class=*"com.sx.domain.Book"* init-method=*"init"* destroy-method=*"destory"*>  <constructor-arg name=*"name"* value=*"F22"*></constructor-arg>  </bean> |

方法

|  |
| --- |
| **public** **void** init() {  System.***out***.println("执行初始化方法");    }  **public** **void** destory() {  System.***out***.println("执行销毁方法");  } |

结果

|  |
| --- |
| 执行初始化方法  Book [name=null, author=null] |

注：只有容器关闭的时候，销毁方法才会结束。

### 引用外部属性文件

数据库连接池作为单实例是最好的；一个项目就一个连接池，连接池里面管理很多连接。连接直接从连接池中拿。

可以让Spring帮我们创建连接池对象，（管理连接池）

配置文件：

|  |
| --- |
| db.driverClassName=com.mysql.jdbc.Driver  db.url=jdbc:mysql://192.168.10.136:3306/ego  db.username=root  db.password=123456 |
| //加载配置文件内容  <context:property-placeholder location="classpath:properties/\*.properties"/>  <bean id="dataSource"  class="com.alibaba.druid.pool.DruidDataSource">  // username与系统的用户名冲突，所以在db.properties写时，加个前缀  <property name="username" value="${db.username}"></property>  <property name="password" value="${db.password}"></property>  <property name="url" value="${db.url}"></property>  <property name="driverClassName"  value="${db.driverClassName}"></property>  </bean> |

注：

1、使用${}获取properties中的文件

2、${}当前后不要有空格，否在会发生字符串拼接。

### 自动装配

按照某种规则自动装配

autowire="byName":按照名字；

private Car car；

以属性名（car）作为id去容器中找到这个组件，给他赋值；如果找不到就装配nu11；

car=ioc.getBean("car");

autowire="byType":以属性的类型作为查找依据去容器中找到这个组件，如果容器中有多类这个类型的组件，会报错，如果没找到，装配null。

private Car car；

car=ioc.getBean(Car.class);

autowire="constructor":按照构造器进行赋值。

1、先按照有参构造的参数类型，进行装配。

如果类型有多个，则会按照与相同参数名进行装配，如果没有相同参数名，贼为null。

2、如果没有就赋值null。

按照类型注入：

|  |
| --- |
| <bean id=*"book"* class=*"com.sx.domain.Book"* >  <property name=*"author"* value=*"sx"*></property>  <property name=*"name"* value=*"ss"*></property>  </bean>  //会根据类型制动找到容器中属性类型是Book类型  <bean id=*"person2"* class=*"com.sx.domain.User"* autowire=*"byType"* >  <property name=*"gender"* value=*"nv"*></property>  </bean> |

结果：

User [name=null, age=null, gender=nv, book=Book [name=ss, author=sx]]

## Spring实例化Bean的三种方式

在spring中，通过配置文件创建对象就叫做Bean实例化，Bean实例化一共3种方式：

### 默认构造器

xml中

|  |
| --- |
| <bean id="" class=""> |

注意：如果提供了id和calss属性，则默认为默认构造方式，必须为class类中提供一个无参的构造器，如果类中没有无参构造会产生异常。

### 用静态工厂创建作用：常用与spring整合其他框架（工具）。

静态工厂：工厂本身不创建对象，通过静态方法调用，对象=工厂类.工厂方法名()得到。

静态工厂:

|  |
| --- |
| **public** **class** AirPlanStaticFactory {  **public** **static** AirPlan getAirPlan(String name) {    AirPlan airPlan = **new** AirPlan();  airPlan.setName(name);  airPlan.setLengh(200);  airPlan.setLengh(200);  **return** airPlan;  }  } |

配置：

|  |
| --- |
| <!--getAirPlan:指定工厂方法 -->  <!--constructor-arg:为参数赋值 -->  <bean id=*"airPlanStaticFactory"* class=*"com.sx.domain.AirPlanStaticFactory"* factory-method=*"getAirPlan"*>  <constructor-arg name=*"name"* value=*"J20"*></constructor-arg>  </bean> |

记过返回的是AirPlan对象。

### 用实例工厂创建UserFactory

实例工厂：必须先有工厂实例对象，通过实例对象创建对象。提供所有的方法都是“非静态”的。

实例工厂：工厂类创建需要对象，

工厂类 = new 工厂类();

工厂对象.getAirPlane(“张三”)

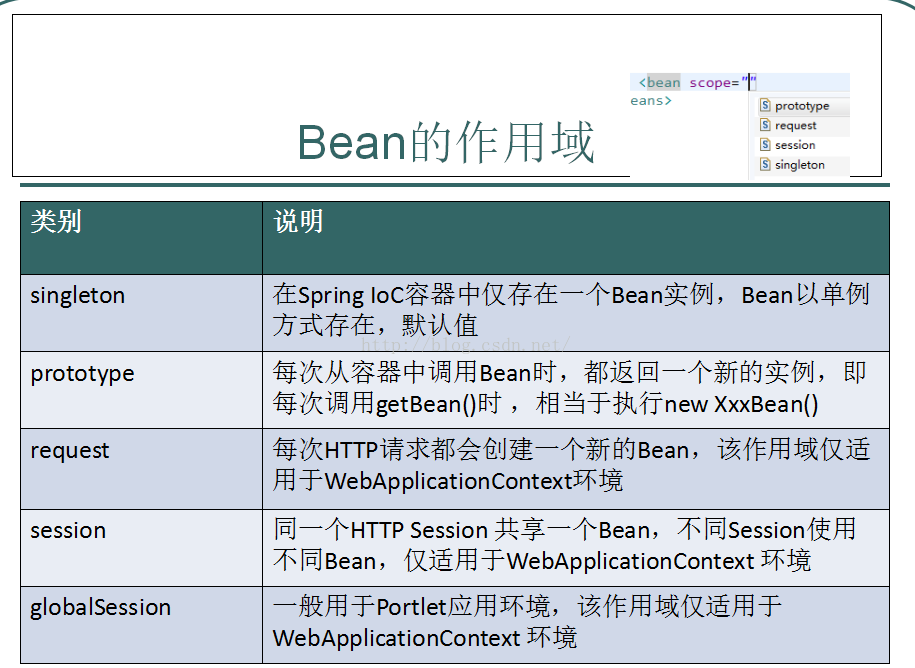
实例工厂

|  |
| --- |
| **public** **class** AirPlanFactory {  **public** AirPlan getAirPlan(String name) {    AirPlan airPlan = **new** AirPlan();  airPlan.setName(name);  airPlan.setLengh(200);  airPlan.setLengh(200);  **return** airPlan;  }  } |

配置

|  |
| --- |
| <!--实例工厂 -->  <!--1、先创建实例工厂对象 -->  <bean id=*"airPlanFactory"* class=*"com.sx.domain.AirPlanFactory"* ></bean>  <!--2、配置要创建的对象，factory-bean指定使用哪个工厂,factory-method指定使用哪个工厂方法 -->  <bean id=*"airPlan"* class=*"com.sx.domain.AirPlan"* factory-bean=*"airPlanFactory"* factory-method=*"getAirPlan"*>  <constructor-arg name=*"name"* value=*"F22"*></constructor-arg>  </bean> |

## 作用域



五种作用域中，request、session和global session三种作用域仅在基于web的应用中使用（不必关心你所采用的是什么web应用框架），只能用在基于web的Spring ApplicationContext环境。

### Singleton(单实例，默认)

当一个bean的作用域为Singleton，那么Spring IoC容器中只会存在一个共享的bean实例，并且所有对bean的请求，只要id与该bean定义相匹配，则只会返回bean的同一实例。Singleton是单例类型，就是在创建起容器时就同时自动创建了一个bean的对象，不管你是否使用，他都存在了，每次获取到的对象都是同一个对象。注意，Singleton作用域是Spring中的缺省作用域。要在XML中将bean定义成singleton，可以这样配置：

|  |
| --- |
| <bean id="serviceImpl" class="cn.csdn.service.ServiceImpl" scope="singleton"> |

注：无论什么时候获取serviceImpl，都是同一个对象，引用是相同的。

### Prototype（多实例）

当一个bean的作用域为Prototype，表示一个bean定义对应多个对象实例。Prototype作用域的bean会导致在每次对该bean请求（将其注入到另一个bean中，或者以程序的方式调用容器的getBean()方法）时都会创建一个新的bean实例。Prototype是原型类型，它在我们创建容器的时候并没有实例化，而是当我们获取bean的时候才会去创建一个对象，而且我们每次获取到的对象都不是同一个对象。根据经验，对有状态的bean应该使用prototype作用域，而对无状态的bean则应该使用singleton作用域。在XML中将bean定义成prototype，可以这样配置：

|  |
| --- |
| <bean id="account" class="com.foo.DefaultAccount" scope="prototype"/>  或者  <bean id="account" class="com.foo.DefaultAccount" singleton="false"/> |

注：每次获取都会创建新的对象，他们的引用不相同。

### Request(web环境中)

当一个bean的作用域为Request，表示在一次HTTP请求中，一个bean定义对应一个实例；即每个HTTP请求都会有各自的bean实例，它们依据某个bean定义创建而成。该作用域仅在基于web的Spring ApplicationContext情形下有效。考虑下面bean定义

|  |
| --- |
| <bean id="loginAction" class=cn.csdn.LoginAction" scope="request"/> |

　针对每次HTTP请求，Spring容器会根据loginAction bean的定义创建一个全新的LoginAction bean实例，且该loginAction bean实例仅在当前HTTP request内有效，因此可以根据需要放心的更改所建实例的内部状态，而其他请求中根据loginAction bean定义创建的实例，将不会看到这些特定于某个请求的状态变化。当处理请求结束，request作用域的bean实例将被销毁。

### Session（同一次会话）

当一个bean的作用域为Session，表示在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。考虑下面bean定义：

|  |
| --- |
| <bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/> |

针对某个HTTP Session，Spring容器会根据userPreferences bean定义创建一个全新的userPreferences bean实例，且该userPreferences bean仅在当前HTTP Session内有效。与request作用域一样，可以根据需要放心的更改所创建实例的内部状态，而别的HTTP Session中根据userPreferences创建的实例，将不会看到这些特定于某个HTTP Session的状态变化。当HTTP Session最终被废弃的时候，在该HTTP Session作用域内的bean也会被废弃掉。

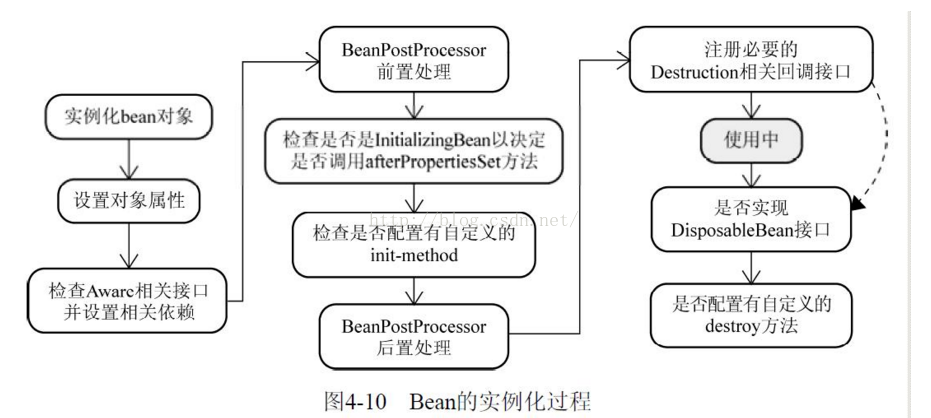
### Global Session

当一个bean的作用域为Global Session，表示在一个全局的HTTP Session中，一个bean定义对应一个实例。典型情况下，仅在使用portlet context的时候有效。该作用域仅在基于web的Spring ApplicationContext情形下有效。考虑下面bean定义：

|  |
| --- |
| <bean id="user" class="com.foo.Preferences "scope="globalSession"/> |

global session作用域类似于标准的HTTP Session作用域，不过仅仅在基于portlet的web应用中才有意义。Portlet规范定义了全局Session的概念，它被所有构成某个portlet web应用的各种不同的portlet所共享。在global session作用域中定义的bean被限定于全局portlet Session的生命周期范围内。

## bean的生命周期



### 初始化和销毁

目标方法执行前后执行后，将进行初始化或销毁。

|  |
| --- |
| <bean id="" class="" init-method="初始化方法名称" destroy-method="销毁的方法名称"> |

对于普通的Java对象，当new的时候创建对象，当它没有任何引用的时候被垃圾回收机制回收。而由Spring IoC容器托管的对象，它们的生命周期完全由容器控制。Spring中每个Bean的生命周期如下：



### 步骤

1实例化bean对象(通过构造方法或者工厂方法)

2设置对象属性(setter等)（依赖注入）

3如果Bean实现了BeanNameAware接口，工厂调用Bean的setBeanName()方法传递Bean的ID。（和下面的一条均属于检查Aware接口）

4如果Bean实现了BeanFactoryAware接口，工厂调用setBeanFactory()方法传入工厂自身

5将Bean实例传递给Bean的前置处理器的postProcessBeforeInitialization(Object bean, String beanname)方法

6调用Bean的初始化方法

7将Bean实例传递给Bean的后置处理器的postProcessAfterInitialization(Object bean, String beanname)方法

8使用Bean

9容器关闭之前，调用Bean的销毁方法

## bean的属性注入（xml方式）（属性赋值）

### 注入方式

1、setter方法注入

提供的propert标签会使用setter方法注入

2.使用有参构造方法注入

提供的有参构造器，构造器会被调用。

3.接口注入【了解】

注：

1、如果在一个bean中定义个另一个bean，这个bean不是引用外部的，这个内部bean，将无法被获取，即使有id。

### set方法

前提：必须提供属性的setter方法。

|  |
| --- |
| <!-- setter方法注入  \* 普通数据  <property name="" value="值">  等效  <property name="">  <value>值  \* 引用数据  <property name="" ref="另一个bean">  等效  <property name="">  <ref bean="另一个bean"/>  -->  <bean id=*"personId"* class=*"com.itheima.f\_xml.b\_setter.Person"*>  //基本类型数据。  <property name=*"pname"* value=*"阳志"*></property>  <property name=*"age"*>  <value>1234</value>  </property>  //引用数据，使用ref标签应用另一个bean的id  <property name=*"homeAddr"* ref=*"homeAddrId"*></property>  <property name=*"companyAddr"*>  <ref bean=*"companyAddrId"*/>  </property>  </bean>    <bean id=*"homeAddrId"* class=*"com.itheima.f\_xml.b\_setter.Address"*>  <property name=*"addr"* value=*"阜南"*></property>  <property name=*"tel"* value=*"911"*></property>  </bean>  <bean id=*"companyAddrId"* class=*"com.itheima.f\_xml.b\_setter.Address"*>  <property name=*"addr"* value=*"北京八宝山"*></property>  <property name=*"tel"* value=*"120"*></property>  </bean>  如果想要将一个属性赋值为null：  <bean id=*"companyAddrId"* class=*"com.itheima.f\_xml.b\_setter.Address"*>  <property name=*"addr"* value=*"null "*></property>//这种方式不可取，此时addr的值为字符串”null”  <property name=*"addr"*>  <null/> //使用property的null标签赋值为null  </property>  <property name=*"tel"* value=*"120"*></property>  </bean> |

### 构造器注入

实体类

|  |
| --- |
| 实体类提供的构造方法  public User(Integer id, String username) {  this.id = id;  this.username = username;  } |

配置：<constructor>的个数要与类中的构造器参数数量相同，否则会报错。

|  |
| --- |
| 1、<!--基于 name 配置构造参数，不常用-->  <!-- <bean id="user" class="com.ittianyu.spring.a\_base.f\_xml.a\_constructor.User">  <constructor -arg name="username" value="name"/>  <constructor-arg name="age" value="1"/>  </bean>-->  2、<!--基于 index 和 type 方式，如果不指定 type，可能匹配到多个符合条件的构造方法，默认使用位置在前面的那个-->  <bean id="User" class="com.ittianyu.spring.a\_base.f\_xml.a\_constructor.User">  <constructor-arg index="0" type="java.lang.Integer" value="1"/>  <constructor-arg index="1" type="java.lang.String" value="hello"/>  </bean>  3、也可以省略name属性，直接写value，但是参数的顺序必须是构造器的顺序。  <bean id="User" class="com.ittianyu.spring.a\_base.f\_xml.a\_constructor.User">  <constructor-arg value="1"/>  <constructor-arg value="hello"/>  </bean>  注：这种方式会参数一个问题，如果有是有两个重载的构造器，spring无法识别到是哪个构造器，会出现随机情况，可以使用type来指定构造器。 例如：  <bean id="User" class="com.ittianyu.spring.a\_base.f\_xml.a\_constructor.User">  <constructor-arg value="1" type="java.lang.Integer"/>  <constructor-arg value="hello"/>  </bean> |

### 集合和数组属性注入

数组：

|  |
| --- |
| <!-- 给citys集合属性赋值，list标签里面指定每个元素，然后再把集合赋给citys -->  <property name="citys">  <array>  //数组中也可以有对象，  普通数据：<value>  引用数据：<ref>  <value>武汉</value>  <value>上海</value>  </array>  </property> |

list

|  |
| --- |
| <!-- 给userList集合属性赋值 -->  <property name="userList">  <list>  <!—1、使用ref的bean属性引用ioc容器中的其它bean对象 -->  <ref bean="user" />  <ref bean="user2" />  <！2、可以在内部创建一个bean>  <bean id=”user3” class=”com.sx.User” ></bean>  </list>  </property> |

map

|  |
| --- |
| <!-- 给maps属性赋值 -->  <property name="maps">  <map>  <entry key="aa" value="上海" />  <entry key="bb" value="武汉" />  <entry key="cc" value="帝都" />  <entry key="cc" /> //注：这个内部bean无法被获取到。  <bean class=”com.sx.User” ></bean>  < /entry>  </map>  </property>  获取map  Animal a = (Animal) ctx.getBean("anim");  Map<String, Object> maps = a.getMaps();  Set<Entry<String, Object>> entrySet = maps.entrySet();  **for** (Entry<String, Object> entry : entrySet) {  System.*out*.println("key:" + entry.getKey() + ",value:"  + entry.getValue()); |

prop(重要)

|  |
| --- |
| <!-- 给props属性赋值，使用props标签配置Properties类型的属性 -->  <property name="props">  <props>  <prop key="driver">com.mysql.jdbc.Driver</prop>  <prop key="username">root</prop>  </props>  </property>  获取props  // 1.获取spring执行环境，加载主配置文件  ApplicationContext ctx = **new** ClassPathXmlApplicationContext(  "classpath:applicationContext.xml");  // 2.从ioc容器中获取bean对象  Animal a = (Animal) ctx.getBean("anim");  Properties p = a.getProps();  String driver = p.getProperty("driver");  String username = p.getProperty("username");  System.*out*.println(driver+"...."+username); |

### 级联属性的赋值

|  |
| --- |
| <bean id=*"companyAddrId"* class=*"com.itheima.f\_xml.b\_setter.Address"*>  <property name=*"user"* ref=*"user1"*></property>  <property name=*"user.name"* value=*"sx"*></property>  </bean>  级联属性赋值：这里使用*user.name 给引用的user1中属性值为name修改为sx，此时引用的user1的name属性也会发生改变。* |

### P标签（P名称空间）

作用：名称空间用来防止标签的重复

|  |
| --- |
| <book>  <b:name>  西游记  </b:name>  <price>19.98</price>  <author>  <a:name>  吴承恩  </a:name>  <gender>男</gender>  </author>  </book> |

例如：上方有两个name，可以使用名称空间区分是book下的name，还是author的name，而不用判断他的父标签。

对“setter方法注入”进行简化，替换<property name="属性名">，而是在 <bean p:属性名="普通值" p:属性名-ref="引用值">

p命名空间使用前提，必须添加命名空间

对setter注入简化

|  |
| --- |
| <bean p:属性名="普通值" p:属性名-ref="引用值">  替换  <property name="" value=""/>  xmlns:xml name space  1、添加命名空间  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:p=<http://www.springframework.org/schema/p>  //复制一份，并添加：p，末尾将beans改为p  xsi:schemaLocation="  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  <!--  利用 setter 方法进行参数注入  在上面要加入 xmlns:p="http://www.springframework.org/schema/p" （从 beans 行复制修改而来）  使用 p 简化 setter 方法  -->  <bean id="Person" class="com.ittianyu.spring.a\_base.f\_xml.b\_setter.Person"  //基本类型  p:name="Jone" p:age="18"//属性=属性值  //引用类型  p:oldCar-ref="OldCar" p:newCar-ref="NewCar">  </bean>  <bean id="OldCar" class="com.ittianyu.spring.a\_base.f\_xml.b\_setter.Car"  p:name="拖拉机" p:price="5000.0">  </bean>  <bean id="NewCar" class="com.ittianyu.spring.a\_base.f\_xml.b\_setter.Car"  p:name="宾利" p:price="5000000.0">  </bean> |

### spEL

对 <property> 进行统一编程，所有的内容都使用 value

<property name="" value="#{表达式}">

#{123}、#{'jack'}： 数字、字符串

#{beanId}：另一个bean引用

#{beanId.propName}：操作数据

#{beanId.toString()}：执行方法

#{T(类).字段|方法}：静态方法或字段

|  |
| --- |
| <!--  <property name="cname" value="#{'jack'}"></property>  <property name="cname" value="#{customerId.cname.toUpperCase()}"></property>  通过另一个bean，获得属性，调用的方法  <property name="cname" value="#{customerId.cname?.toUpperCase()}"></property>  ?. 如果对象不为null，将调用方法  -->  <bean id=*"customerId"* class=*"com.itheima.f\_xml.d\_spel.Customer"* >  <property name=*"cname"* value=*"#{customerId.cname?.toUpperCase()}"*></property>  <property name=*"pi"* value=*"#{T(java.lang.Math).PI}"*></property>  </bean> |

## 装配bean

### Spring提供了三种装配bean的方式

1、在XML中进行显示配置；

2、在Java中进行显示配置；

3、隐式的bean发现机制和自动装配；

注：三种装配方式可以结合使用，但是推荐首选第3种自动装配，之后选用Java进行装配，最后选用XML进行装配。

## 重构三层结构项目

### 普通版

|  |
| --- |
| <!-- 实例化UserDao -->  <bean id=*"userDao"* class=*"cn.sxt.dao.impl.UserDaoImpl2"*> </bean>    <!-- 实例化UserService -->  <bean id=*"userService"* class=*"cn.sxt.service.impl.UserServiceImpl"*>  <property name=*"userDao"* ref=*"userDao"*></property>  </bean>    <!-- 实例化UserController -->  <bean id=*"userController"* class=*"cn.sxt.controller.UserController"*  autowire=*"byType"*>  <property name=*"service"* ref=*"userService"*></property>  </bean> |

### 进阶版

自动装配属性:autowire，使用自动装配可以省略set注入。

|  |
| --- |
| <!-- 实例化UserDao -->  <bean id=*"userDao"* class=*"cn.sxt.dao.impl.UserDaoImpl2"*>  </bean>  <!-- 实例化UserService -->  <bean id=*"userService"* class=*"cn.sxt.service.impl.UserServiceImpl"*  autowire=*"byName"*>//  </bean>    <!-- 实例化UserController -->  <bean id=*"userController"* class=*"cn.sxt.controller.UserController"*  autowire=*"byType"*>  </bean> |

## 注解注入bean

注解 ：参与代码编译，以@开头的。它是给应用程序看的，单独使用注解毫无意义，一定要跟工具一起使用,这个所谓的工具实际就是能读懂注解的应用程序

如果 Web 应用程序采用了经典的三层分层结构的话，最好在持久层、业务层和控制层分别采用 @Repository、@Service 和 @Controller 对分层中的类进行注释，而用 @Component 对那些比较中立的类进行注释。

在一个稍大的项目中，通常会有上百个组件，如果这些组件采用xml的bean定义来配置，显然会增加配置文件的体积，查找以及维护起来也不太方便。 Spring2.5为我们引入了组件自动扫描机制，他可以在类路径底下寻找标注了@Component,@Service,@Controller,@Repository注解的类，并把这些类纳入进Spring容器中管理。它的作用和在xml文件中使用bean节点配置组件时一样的

主要目的：取代xml

使用注解加入到容器中的组件，和使用配置加入到容器中的组件行为都是默认一样的;

1、组件的id。默认就是组件的类名首字母小写;

2、组件的作用域，默认就是单例的;

通过给bean上添加某些注解，可以快速的将bean加入到ioc容器中，某个类上添加上任何一个注解都能快速的将这个组件加入到ioc容器的管理中；

Spring有四个注解：

@Controller:控制器；我们推荐给控制器层（servlet包下的这些）的组件加这个注解

@Service:业务逻辑；我们推荐业务逻辑层的组件添加这个注解；BookService

@Repository:给数据库层（持久化层，dao层）的组件添加这个注解

@Component:给不属于以上几层的组件添加这个注解；

注解可以随便加；Spring底层不会去验证你的这个组件，是否如你注解所说就是一个dao层的或者就是一个servlet层的组件；我们推荐各自层加各自注解；给我们程序员看的；

使用注解将组件快速的加入到容器中需要几步；

1）、给要添加的组件上标四个注解的任何一个

2）、告诉Spring，自动扫描加了注解的组件；依赖context名称空间

context：component-scan:自动组件扫描

**base**-package:指定扫描的基础包；把基础包及他下面所有的包的所有加了注解的类，自动的扫描进ioc容器。

3）、一定要导入aop包，支持加注解模式的；

### 通过注解装配bean

pojo

|  |
| --- |
| @Component(value="user")  public class User {  @Value("sx")  private String name;  @Value("22")  private Integer age;  //getter和setter方法  } |

注解@Component代表Spring IoC会把这个类扫描生成Bean实例，而其中的value属性代表这个类在Spring中的id，这就相当于XML方式定义的Bean的id，也可以简写成@Component（“user")，甚至直接写成@Component，对于不写的，Spring loC容器就默认类名，但是以首字母小写的形式作为id，为其生成对象，配置到容器中。

注解@Value代表的是值的注入，这里只是简单注入一些值，其中id是一个long型，注入的时候Spring会为其转化类型。

现在有了这个类，但是还不能进行测试，因为Spring IoC 并不知道需要去哪里扫描对象，这个时候可以使用一个Java Config 来去告诉它，

Java Config 类

|  |
| --- |
| **import** org.springframework.context.annotation.ComponentScan;  @ComponentScan  **public** **class** PojoConfig {  } |

@ComponentScan 代表进行扫描， 默认是扫描当前包的路径， POJO 的包名和它保持一致才能扫描，否则是没有的。

测试

|  |
| --- |
| @Test  **public** **void** test(){  ApplicationContext applicationContext =  **new** AnnotationConfigApplicationContext(PojoConfig.**class**);  User user = applicationContext.getBean(User.**class**);  System.*out*.println(user);// User [name=sx, age=22]  } |

这里可以看到使用了AnnotationConfigApplicationContext类去初始化Spring IoC容器，

这样Spring IoC就会根据注解的配置去解析对应的资源，来生成loC容器了。

由此可以看到两个明显的弊端： 其一， 对于＠ComponentScan 注解， 它只是扫描所在包的Java 类， 但是更多的时候真正需要的是可以扫描所指定的类； 其二， 上面只注入了一些简单的值，而没有注入对象，同样在现实的开发中可以注入对象是十分重要的，也是常见的场景。

### 自动装配

下列三个都是自动装配

@Autowired ：最强大，spring自己的注解。

@Resource :j2ee，JSR250规范实现的

@Inject, JSR330规范实现的

@Resource:拓展性强,如果切换成另一种框架，@Resource还是可以用的,@ Autowired不行

### ＠Autowired 自动装配

@ Autowired注入原理：

1、先按照类型去找对应的组件：

1)找到一个：赋值

2)没有找到：报异常

3)找到多个：可能有子类继承父类，子类对象和父类对象是同一类型。

1)按照变量的id继续匹配bean。

匹配到：

没有匹配到：报错，解决方法：使用@Qualifier(“变量名”)

总结：@ Autowired一定会找到，注入。找不到，报异常。如果想要设置找不到设置null，可以使用@ Autowired(required=false)

通过学习Spring IoC 容器，我们知道Spring 是先完成Bean 的定义和生成，然后寻找需要注入的资源。也就是当Spring 生成所有的Bean 后，如果发现这个注解，它就会在Bean中查找，然后找到对应的类型，将其注入进来，这样就完成依赖注入了。所谓自动装配技术是一种由Spring 自己发现对应的Bean ， 自动完成装配工作的方式，它会应用到一个十分常用的注解＠Autowired 上， 这个时候Spring 会根据类型去寻找定义的Bean 然后将其注入，这里需要留意按类型(Role)的方式。

|  |
| --- |
| @Component  **public** **class** UserServiceImpl {  @Autowired  User user;  } |

s

|  |
| --- |
| @Test  **public** **void** test(){  ApplicationContext applicationContext =  **new** AnnotationConfigApplicationContext(PojoConfig.**class**);  UserServiceImpl impl = applicationContext.getBean(UserServiceImpl.**class**);  impl.info();  } |

这里的＠Autowired 注解，表示在Spring IoC 定位所有的Bean 后，这个字段需要按类型注入，这样IoC 容器就会寻找资源，然后将其注入。

那么Spring IoC 容器会为它们先生成对应的实例，然后依据＠Auto wired 注解，按照类型找到定义的实例，将其注入。

注入IoC 容器有时候会寻找失败，在默认的情况下寻找失败它就会抛出异常，也就是说默认情况下， S pring IoC 容器会认为一定要找到对应的Bean 来注入这个字段，有些时候这并不是一个真实的需要， 比如日志，有时候我们会觉得这是可有可无的，这个时候可以通过@A utowired 的配置项required 来改变它，比如＠ Autowired(required =false） 。

正如之前所谈到的在默认情况下是必须注入成功的，所以这里的required 的默认值为true 。当把配置修改为了false 时，就告诉Spring IoC 容器，假如在已经定义好的Bean 中找不到对应的类型，允许不注入，这样也就没有了异常抛出，只是这样这个字段可能为空，读者要自行校验，以避免发生空指针异常。在大部分的情况下，都不需要这样修改。

方法上有@Autowired的化；

1）、这个方法也会在bean创建的时候自动运行

2）、这个方法上的每一个参数都会自动注入值

3）、方法的参数上也可以标注解，

@Autowired 除可以配置在属性之外，还允许方法配置，常见的Bean 的setter 方法也可以使用它来完成注入。

|  |
| --- |
| @Autowired  public void setUser( @Qulifier(“user1”)User user) {  this.user = user;  } |

### 自动装配的歧义性

注解@Primary代表首要的，当Spring IoC通过一个接口或者抽象类注入对象的时候，

由于存在多个实现类或者具体类，就会犯糊涂，不知道采用哪个类注入为好。注解@Primary则是告诉Spring loC容器，请优先使用该类注入。

|  |
| --- |
| @Component  @Primary  **public** **class** UserServiceImpl {  @Autowired  User user;  } |

### Controller

@Controller:标注一个控制器组件类。对应表现层的Bean，也就是Action。

代码

|  |
| --- |
| @Controller  @Scope("prototype")  public class UserAction extends BaseAction<User>{  ……  } |

使用@Controller注解标识UserAction之后，就表示要把UserAction交给Spring容器管理，在Spring容器中会存在一个名字为"userAction"的action，这个名字是根据UserAction类名来取的。

注意：如果@Controller不指定其value【@Controller】，则默认的bean名字为这个类的类名首字母小写，如果指定value【@Controller(value="UserAction")】或者【@Controller("UserAction")】，则使用value作为bean的名字。

这里的UserAction还使用了@Scope注解，@Scope("prototype")表示将Action的范围声明为原型，可以利用容器的scope="prototype"来保证每一个请求有一个单独的Action来处理，避免struts中Action的线程安全问题。spring 默认scope 是单例模式(scope="singleton")，这样只会创建一个Action对象，每次访问都是同一Action对象，数据不安全，struts2 是要求每次次访问都对应不同的Action，scope="prototype" 可以保证当有请求的时候都创建一个Action对象。

### Service

@Service对应的是业务层Bean，例如：

|  |
| --- |
| @Service("userService")  public class UserServiceImpl implements UserService {  ………  } |

@Service("userService")注解是告诉Spring，当Spring要创建UserServiceImpl的的实例时，bean的名字必须叫做"userService"，这样当Action需要使用UserServiceImpl的的实例时,就可以由Spring创建好的"userService"，然后注入给Action：在Action只需要声明一个名字叫“userService”的变量来接收由Spring注入的"userService"即可，具体代码如下：

|  |
| --- |
| // 注入userService  @Resource(name = "userService")  private UserService userService; |

3 private UserService userService;

注意：在Action声明的“userService”变量的类型必须是“UserServiceImpl”或者是其父类“UserService”，否则由于类型不一致而无法注入，由于Action中的声明的“userService”变量使用了@Resource注解去标注，并且指明了其name = "userService"，这就等于告诉Spring，说我Action要实例化一个“userService”，你Spring快点帮我实例化好，然后给我，当Spring看到userService变量上的@Resource的注解时，根据其指明的name属性可以知道，Action中需要用到一个UserServiceImpl的实例，此时Spring就会把自己创建好的名字叫做"userService"的UserServiceImpl的实例注入给Action中的“userService”变量，帮助Action完成userService的实例化，这样在Action中就不用通过“UserService userService = new UserServiceImpl();”这种最原始的方式去实例化userService了。如果没有Spring，那么当Action需要使用UserServiceImpl时，必须通过“UserService userService = new UserServiceImpl();”主动去创建实例对象，但使用了Spring之后，Action要使用UserServiceImpl时，就不用主动去创建UserServiceImpl的实例了，创建UserServiceImpl实例已经交给Spring来做了，Spring把创建好的UserServiceImpl实例给Action，Action拿到就可以直接用了。Action由原来的主动创建UserServiceImpl实例后就可以马上使用，变成了被动等待由Spring创建好UserServiceImpl实例之后再注入给Action，Action才能够使用。这说明Action对“UserServiceImpl”类的“控制权”已经被“反转”了，原来主动权在自己手上，自己要使用“UserServiceImpl”类的实例，自己主动去new一个出来马上就可以使用了，但现在自己不能主动去new“UserServiceImpl”类的实例，new“UserServiceImpl”类的实例的权力已经被Spring拿走了，只有Spring才能够new“UserServiceImpl”类的实例，而Action只能等Spring创建好“UserServiceImpl”类的实例后，再“恳求”Spring把创建好的“UserServiceImpl”类的实例给他，这样他才能够使用“UserServiceImpl”，这就是Spring核心思想“控制反转”，也叫“依赖注入”，“依赖注入”也很好理解，Action需要使用UserServiceImpl干活，那么就是对UserServiceImpl产生了依赖，Spring把Acion需要依赖的UserServiceImpl注入(也就是“给”)给Action，这就是所谓的“依赖注入”。对Action而言，Action依赖什么东西，就请求Spring注入给他，对Spring而言，Action需要什么，Spring就主动注入给他。

### @Repository

@Repository对应数据访问层Bean ，例如：

|  |
| --- |
| @Repository(value="userDao")  public class UserDaoImpl extends BaseDaoImpl<User> {  ………  } |

@Repository(value="userDao")注解是告诉Spring，让Spring创建一个名字叫“userDao”的UserDaoImpl实例。

当Service需要使用Spring创建的名字叫“userDao”的UserDaoImpl实例时，就可以使用@Resource(name = "userDao")注解告诉Spring，Spring把创建好的userDao注入给Service即可。

|  |
| --- |
| // 注入userDao，从数据库中根据用户Id取出指定用户时需要用到  @Resource(name = "userDao")  private BaseDao<User> userDao; |

注：

1、使用注解产生的bean对象id默认为类名首字母小写形式。

2、注解之后可以添加别名，相当于在xml文件中设置的id。

### 依赖注入

由spring的ioc创建好对象之后，需要将对象注入到对应的声明处。

私有字段设置，也可以给setter方法设置

普通值：@Value("")

引用值：

方式1：按照【类型】注入

@Autowired

注：这种方式不推荐，一个接口可能有两个实现类，ioc不知道选择使用哪一个。

方式2：按照【名称】注入1

@Autowired

@Qualifier("名称")

方式3：按照【名称】注入2

@Resource("名称")

@Resource是JSR250规范的实现，需要导入javax.annotation实现注入。

@Resource是根据名称进行自动装配的，一般会指定一个name属性

@Resource可以作用在变量、setter方法上。

4.生命周期

初始化：@PostConstruct

销毁：@PreDestroy

### 多实例

@Scope("prototype") 多例

### 使用注解前添加命名空间

appliactionContext.xml

|  |
| --- |
| <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context.xsd"> |

### 在xml文件开启扫描组件

注解使用前提，添加命名空间，让spring扫描含有注解类

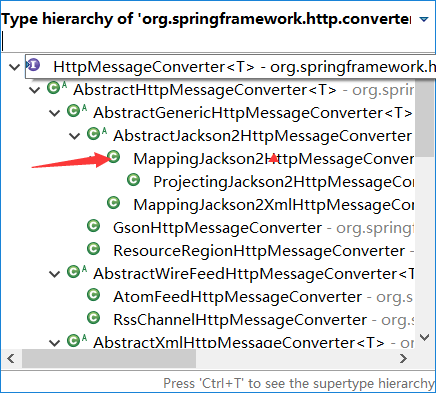
|  |
| --- |
| <!-- 开启组件扫描 base-package:要扫描的包,spring会扫描这些包下的类 -->  <!-- 扫描Dao -->  <context:component-scan base-package=*"cn.sxt.dao.impl"*></context:component-scan>  <!-- 扫描Service -->  <context:component-scan base-package=*"cn.sxt.service.impl"*></context:component-scan>  <!-- 扫描controller -->  <context:component-scan base-package=*"cn.sxt.controller"*></context:component-scan>  <!-- 扫描*domain* -->  <context:component-scan base-package=*"cn.sxt.domain"*></context:component-scan>  <!-- 全局扫描配置，即多个package写在一起，用，隔开 -->  <context:component-scan base-package=*"cn.sxt.dao.impl,cn.sxt.service.impl,cn.sxt.controller"*></context:component-scan>  <!-- 全局扫描配置，扫描全部包-->  <context:component-scan base-package="cn.sxt"></context:component-scan> |

### 创建对象

|  |
| --- |
| DaoImpl  @Repository("daoimp") //这里指定bean的id。可以通过该id获取指定的该对象。  **public** **class** AnimalDaoImpl **implements** AnimalDao{  @Override  **public** Animal queryById(Integer id) {  Animal animal = **new** Animal();  }  }  @Service("serviceimpl")//这里指定bean的id。可以通过该id获取指定的该对象。  **public** **class** AnimalServiceImpl **implements** AnimalService {  @Autowired//使用自动类型加载方式，使用该对象时，会自动去ioc容器中寻找。  AnimalDaoImpl animalDaoImpl;  @Override  @ Autowired  @  **public** Animal getAnimalById(Integer id) {  Animal animal = animalDaoImpl.queryById(id);  **return** animal;  }  }  Controller层：  @Controller  **public** **class** AnimalController {  @Autowired  @Qualifier("daoimp")//这里通过指定类型和名字去加载AnimalService这个对象，如果没有daoimp的AnimalService，将无法加载到。  AnimalService animalService;    Animal getAnimal(Integer id){  Animal animal = animalService.getAnimalById(id);  **return** animal;  }  } |

### @requestBody

httpMessageConvert



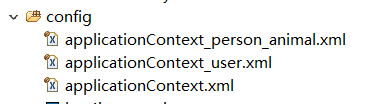
## spring的模块化配置

在开发中，我们一般会分模块去管理我们的spring xml文件，

dao放到app\_dao.xml里面

service放到app\_service.xml里面

action放到app\_action.xml里面



不同配类的配置文件分开。最后在applicationContext.xml中使用import引入

<import resource="classpath:applicationContext\_user.xml"/>

<import resource="classpath:applicationContext\_person\_animal.xml"/>

## 组件扫描

### 自动组件扫描

<context:component-scan base-package=*"com.sx.controller"*></context:component-scan>

指定扫描的基础包；把基础包及他下面所有的包的所有加了注解的类，自动的扫描进ioc容器。

## spring的单元测试

使用Spring的单元测试；

1、导包；Spring单元测试包spring-test-4.e.e.RELEASE.jar

2、@ContextConfiguration(locations="")使用它来指定Spring的配置文件的位置

3、@Runwith指定用哪种驱动进行单元测试，默认就是junit

# AOP(面向切面编程)

## 什么是AOP

### aop概述

在软件业，AOP为Aspect Oriented Programming的缩写，意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是OOP（面向对象编程）的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

特点：AOP采取横向抽取机制，取代了传统纵向继承体系重复性代码

经典应用：事务管理、性能监视、安全检查、缓存 、日志等

性能监视：执行代码的时间。

实现方式：Spring AOP使用纯Java实现，不需要专门的编译过程和类加载器，在运行期通过代理方式向目标类织入增强代码

AspectJ是一个基于Java语言的AOP框架，Spring2.0开始，Spring AOP引入对Aspect的支持，AspectJ扩展了Java语言，提供了一个专门的编译器，在编译时提供横向代码的织入

这种在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面的编程。

核心思想：

1、只有在运行的时候，代码会自动添加到类中。

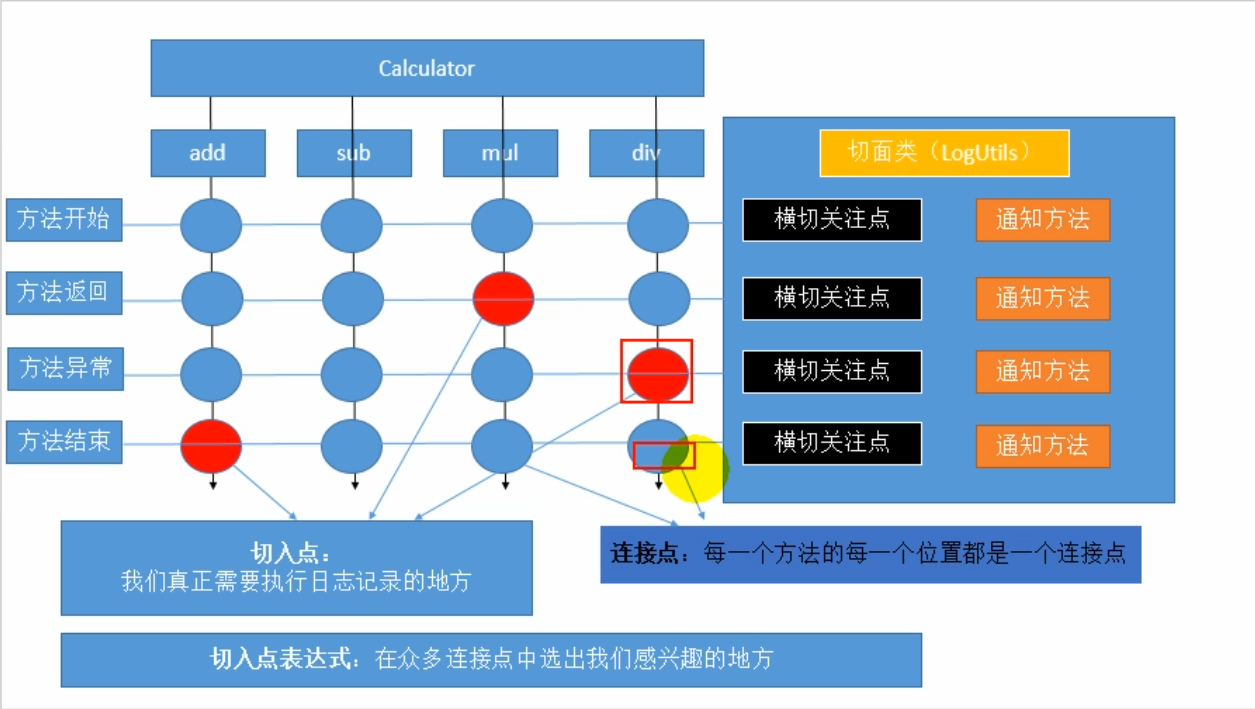
2、不会与类发生关联，即修改代码时不会对原本的类产生影响。

面向切面编程（AOP是Aspect Oriented Program的首字母缩写） ，我们知道，面向对象的特点是继承、多态和封装。而封装就要求将功能分散到不同的对象中去，这在软件设计中往往称为职责分配。实际上也就是说，让不同的类设计不同的方法。这样代码就分散到一个个的类中去了。这样做的好处是降低了代码的复杂程度，使类可重用。 但是人们也发现，在分散代码的同时，也增加了代码的重复性。什么意思呢？比如说，我们在两个类中，可能都需要在每个方法中做日志。按面向对象的设计方法，我们就必须在两个类的方法中都加入日志的内容。也许他们是完全相同的，但就是因为面向对象的设计让类与类之间无法联系，而不能将这些重复的代码统一起来。 也许有人会说，那好办啊，我们可以将这段代码写在一个独立的类独立的方法里，然后再在这两个类中调用。但是，这样一来，这两个类跟我们上面提到的独立的类就有耦合了，它的改变会影响这两个类。那么，有没有什么办法，能让我们在需要的时候，随意地加入代码呢？这种在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面的编程。

一般而言，我们管切入到指定类指定方法的代码片段称为切面，而切入到哪些类、哪些方法则叫切入点。有了AOP，我们就可以把几个类共有的代码，抽取到一个切片中，等到需要时再切入对象中去，从而改变其原有的行为。这样看来，AOP其实只是OOP的补充而已。OOP从横向上区分出一个个的类来，而AOP则从纵向上向对象中加入特定的代码。有了AOP，OOP变得立体了。如果加上时间维度，AOP使OOP由原来的二维变为三维了，由平面变成立体了。从技术上来说，AOP基本上是通过代理机制实现的。

AOP在编程历史上可以说是里程碑式的，对OOP编程是一种十分有益的补充。

### aop相关概念



**1、Target（目标）**：代理对象的目标对象（要增强的类）例如：UserService类

**2、JoinPoint（连接点）**：所谓连接点是指那些可能拦截的点，而spring中这些点就是指方法，因为spring只支持方法类型的连接点。例如：UserService所有的方法。

**3，PointCut（切入点）**：已经被增强的连接点。例如：addUser()方法

**4，Advice（通知/增强）**：所谓通知/增强，就是指拦截到JoinPoint后需要完成的代码。

前置通知/增强，

后置通知/增强，

异常通知/增强，

最终通知/增强，环绕

通知/增强（切面要完成的功能）；

**5，Weaving（织入）**：是指把增强advice应用到目标对象target来创建新的代理对象proxy的过程.

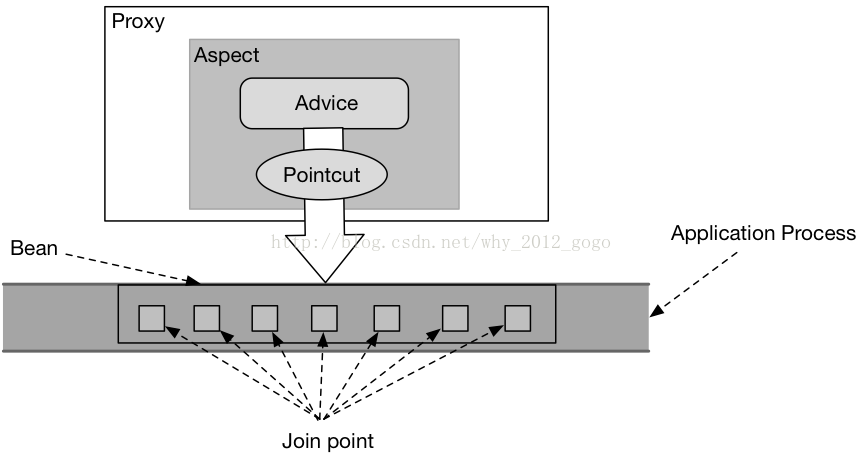
**4，Introduction（引介）**：引介是一种特殊的Advice，在不修改代码的前提下，引介可以在运行期为类动态的添加一些方法或Field。

**7，Proxy（代理）**：一个类被AOP注入增强后，就产生了一个结果代理类。spring

实现了接口：使用jdk动态代理

没有实现接口：使用cglib代理

**8，Aspect（切面）**：是PointCut和Advice（Introduction）的结合





spring的aop编程应用在项目的service层，可以对service里面的业务方法进行增强（前置增强，后置增强，环绕增强异常增强。）

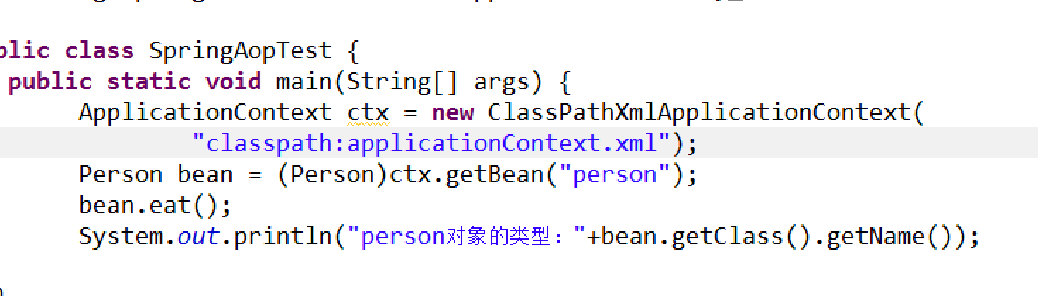
1、对某些业务方法进行日志插入，保证方法不用记日志。

2、管理实务，service调dao时候需要实务，让oap统一管理事务。

3、性能监控，可以写环绕通知获取某个业务方法执行的时间，如果时间过长，需要优化。

环绕通常可以替代前置通知和后置通知。

### 代理模式



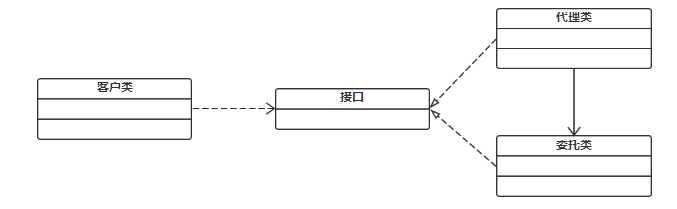
这个是person增强后的代理对象，由cglib创建。

## 代理模式

### 代理模式

代理模式的定义：代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。通俗的来讲代理模式就是我们生活中常见的中介。

举个例子来说明：假如说我现在想买一辆二手车，虽然我可以自己去找车源，做质量检测等一系列的车辆过户流程，但是这确实太浪费我得时间和精力了。我只是想买一辆车而已为什么我还要额外做这么多事呢？于是我就通过中介公司来买车，他们来给我找车源，帮我办理车辆过户流程，我只是负责选择自己喜欢的车，然后付钱就可以了。用图表示如下：



客户只关心买车，买车的手续（例如办证，跑腿）操作代理类来实现，具体买车还是委托类来执行。代理类提供的功能是买车的手续。

### 为什么要使用代理模式

**中介隔离作用**：在某些情况下，一个客户类不想或者不能直接引用一个委托对象，而代理类对象可以在客户类和委托对象之间起到中介的作用，其特征是代理类和委托类实现相同的接口。

**开闭原则，增加功能**：代理类除了是客户类和委托类的中介之外，我们还可以通过给代理类增加额外的功能来扩展委托类的功能，这样做我们只需要修改代理类而不需要再修改委托类，符合代码设计的开闭原则。代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后对返回结果的处理等。代理类本身并不真正实现服务，而是同过调用委托类的相关方法，来提供特定的服务。真正的业务功能还是由委托类来实现，但是可以在业务功能执行的前后加入一些公共的服务。例如我们想给项目加入缓存、日志这些功能，我们就可以使用代理类来完成，而没必要打开已经封装好的委托类。

### 代理模式的种类

我们有多种不同的方式来实现代理。如果按照代理创建的时期来进行分类的话， 可以分为两种：静态代理、动态代理。静态代理是由程序员创建或特定工具自动生成源代码，在对其编译。在程序员运行之前，代理类.class文件就已经被创建了。动态代理是在程序运行时通过反射机制动态创建的。

## 静态代理

### 定义接口（实现的功能）

|  |
| --- |
| public interface BuyHouse {  void buyHosue();  } |

### 委托类实现接口(真正实现购买操作)

|  |
| --- |
| public class BuyHouseImpl implements BuyHouse {  @Override  public void buyHosue() {  System.out.println("我要买房");  }  } |

### 代理类

|  |
| --- |
| public class BuyHouseProxy implements BuyHouse {  //添加委托类，实现对委托类的管理，完成买房的操作对象的控制。  private BuyHouse buyHouse;  //将委托类对象注入  public BuyHouseProxy(final BuyHouse buyHouse) {  this.buyHouse = buyHouse;  }  //定义一些买房前操作和买房后操作。  @Override  public void buyHosue() {  System.out.println("买房前准备");  buyHouse.buyHosue();//执行买房的动作。  System.out.println("买房后装修");  }  } |

### 测试

|  |
| --- |
| public class ProxyTest {  public static void main(String[] args) {  BuyHouse buyHouse = new BuyHouseImpl();//执行买房的对象  buyHouse.buyHosue();  //将买房对象注入到管理类中。管理类对象可以对买房全部过程进行控制。  BuyHouseProxy buyHouseProxy = new BuyHouseProxy(buyHouse);  buyHouseProxy.buyHosue();  }  } |

### 总结

优点：可以做到在符合开闭原则的情况下对目标对象进行功能扩展。

缺点：我们得为每一个服务都得创建代理类，工作量太大，不易管理。同时接口一旦发生改变，代理类也得相应修改。

## JDK的动态代理

JDK 动态代理主要涉及到 java.lang.reflect 包中的两个类：Proxy 和 InvocationHandler。InvocationHandler是一个接口，通过实现该接口定义横切逻辑，并通过反射机制调用目标类的代码，动态将横切逻辑和业务逻辑编制在一起。Proxy 利用 InvocationHandler 动态创建一个符合某一接口的实例，生成目标类的代理对象。

### 定义接口（实现的功能）

|  |
| --- |
| public interface BuyHouse {  void buyHosue();  } |

### 委托类实现接口(真正实现购买操作)

|  |
| --- |
| public class BuyHouseImpl implements BuyHouse {  @Override  public void buyHosue() {  System.out.println("我要买房");  }  } |

### 代理工厂

|  |
| --- |
| **public** **class** ProxyFactory {  //引用目标对象,可以代理任意的目标类  **private** Object target;    **public** ProxyFactory(Object target) {  **super**();  **this**.target = target;  }    //创建代理对象的方法  **public** Object getProxyInstance(){  Object obj=**null**;  //参数1：类加载器  ClassLoader loader=**this**.target.getClass().getClassLoader();  // 参数2：代理类实现的接口列表,代理类实现的接口和目标类相同  Class[] interfaces=**this**.target.getClass().getInterfaces();  //参数3：调用处理程序，当我们调用代理对象的方法时，会自动调用invoke()  InvocationHandler h=**new** InvocationHandler() {  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  //前置增强  System.*out*.println("调用目标方法前的日志：方法名："+method.getName());  //调用目标类的目标方法  Object result = method.invoke(target, args);  //后者增加  System.*out*.println("目标方法调用完毕");  **return** result;  }  };  //调用JDK提供的Proxy.newProxyInstance获取一个代理对象  obj=Proxy.*newProxyInstance*(loader, interfaces, h);  **return** obj;  }    } |

### 测试

|  |
| --- |
| public static void main(String[] args) {  //创建目标对象  IStar star=new Star();  //创建代理工厂  ProxyFactory factory=new ProxyFactory(star);  //产生代理对象，代理对象也实现了与委托类相同的接口  IStar proxy =(IStar) factory.getProxyInstance();    proxy.sing("周杰伦");  //com.sun.proxy.$Proxy0  System.out.println("代理对象的类型名称："+proxy.getClass().getName());  } |

注意Proxy.newProxyInstance()方法接受三个参数：

ClassLoader loader:指定当前目标对象使用的类加载器,获取加载器的方法是固定的

Class<?>[] interfaces:指定目标对象实现的接口的类型,使用泛型方式确认类型

InvocationHandler:指定动态处理器，执行目标对象的方法时,会触发事件处理器的方法

### 总结

虽然相对于静态代理，动态代理大大减少了我们的开发任务，同时减少了对业务接口的依赖，降低了耦合度。但是还是有一点点小小的遗憾之处，那就是它始终无法摆脱仅支持interface代理的桎梏，因为它的设计注定了这个遗憾。回想一下那些动态生成的代理类的继承关系图，它们已经注定有一个共同的父类叫Proxy。Java的继承机制注定了这些动态代理类们无法实现对class的动态代理，原因是多继承在Java中本质上就行不通。有很多条理由，人们可以否定对 class代理的必要性，但是同样有一些理由，相信支持class动态代理会更美好。接口和类的划分，本就不是很明显，只是到了Java中才变得如此的细化。如果只从方法的声明及是否被定义来考量，有一种两者的混合体，它的名字叫抽象类。实现对抽象类的动态代理，相信也有其内在的价值。此外，还有一些历史遗留的类，它们将因为没有实现任何接口而从此与动态代理永世无缘。如此种种，不得不说是一个小小的遗憾。但是，不完美并不等于不伟大，伟大是一种本质，Java动态代理就是佐例。

## Cglib的动态代理

JDK实现动态代理需要实现类通过接口定义业务方法，对于没有接口的类，如何实现动态代理呢，这就需要CGLib了。CGLib采用了非常底层的字节码技术，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。但因为采用的是继承，所以不能对final修饰的类进行代理。JDK动态代理与CGLib动态代理均是实现Spring AOP的基础。

Cglib子类代理实现方法:

1.需要引入cglib的jar文件,但是Spring的核心包中已经包括了Cglib功能,所以直接引入spring-core-3.2.5.jar即可.

2.引入功能包后,就可以在内存中动态构建子类

3.代理的类不能为final,否则报错

4.目标对象的方法如果为final/static,那么就不会被拦截,即不会执行目标对象额外的业务方法.

### 类方法

|  |
| --- |
| **public** **void** sing(String name) {  System.*out*.println(name+"正在唱千里之外!");  } |

### 代理类工厂

|  |
| --- |
| **public** **class** ProxyFactory **implements** MethodInterceptor{  //目标对象  **private** Object target;    **public** ProxyFactory(Object target) {  **super**();  **this**.target = target;  }  //产生代理对象  **public** Object getProxyInstance(){  //工具类  Enhancer ech=**new** Enhancer();  //设置父类为目标类  ech.setSuperclass(**this**.target.getClass());  //设置回调方法,自动调用下面的intercept方法  ech.setCallback(**this**);  **return** ech.create();  }  //intercept方法：拦截到目标对象的方法时，要执行的代码就写在该方法里面  //返回目标方法的返回值  @Override  **public** Object intercept(Object obj, Method method, Object[] args,  MethodProxy arg3) **throws** Throwable {  //前置增强  System.*out*.println("调用目标方法前日志");  //调用目标对象的方法  Object result = method.invoke(**this**.target, args);  //后者增强  System.*out*.println("目标方法执行完毕");  **return** result;  }  } |

## 普通aop开发

 从spring容器获得目标类，如果配置aop，spring将自动生成代理。

 要确定目标类，aspectj 切入点表达式，导入jar包

### 导入jar包

spring-aop-5.1.4.RELEASE.jarz 基本包

aspectjweaver-1.9.2.jar 外部包

实现，不需要接口也可以，实现动态代理。

### 配置文件

步骤：

1、创建目标类

2、创建切面类（通知）

3、aop编程

3.1使用<aop:config>进行配置

<aop：pointcut>切入点，从目标对象获取增强的方法。

expression：切入点表达式，execution(返回类型 包名.类名.方法名(形参)

<aop:advisor>特殊的切面，只有一个通知和一个切入点。

属性：advice-ref：通知的引用。

point-ref：切入点的引用。通过切入点id引用该增强的方法。

切入点表达式：

3.3 切入点表达式

execution(\* com.itheima.c\_spring\_aop.\*.\*(..))

选择方法 返回值任意 包 类名任意 方法名任意 参数任意

|  |
| --- |
| <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd*  *"*>  <!-- 目标类的对象 -->  <bean id=*"person"* class=*"cn.sxt.domain.Person"*></bean>  <!-- 前置通知的对象 -->  <bean id=*"beforeadvice"* class=*"cn.sxt.advice.MyBeforeAdvice"*></bean>  <!-- 后置通知的对象 -->  <bean id=*"afteradvice"* class=*"cn.sxt.advice.MyAfterAdvice"*></bean>  <!-- 环绕通知 -->  <bean id=*"arounddvice"* class=*"cn.sxt.advice.MyAroundAdvice"*></bean>  <!-- 异常通知 -->  <bean id=*"exceptionAdvice"* class=*"cn.sxt.advice.MyExceptionAdvice"*>  <!-- aop的配置 -->  <aop:config>  <!-- 切入点：要增强的方法 expression:切入点表达式 切入点表达式规则： execution(返回类型 包名.类名.方法名(形参)  ) -->  <aop:pointcut expression=*"execution(void cn.sxt.domain.Person.eat(..))"*  id=*"pct"* />  <!-- aop:advisor:完成织入，组合切入点和通知 advice-ref:引用Advice实例,通过id引用 pointcut-ref:引用切入点 -->  <!-- <aop:advisor advice-ref="beforeadvice" pointcut-ref="pct" />  <aop:advisor advice-ref="afteradvice" pointcut-ref="pct" /> -->  <!-- <aop:advisor advice-ref="arounddvice" pointcut-ref="pct" /> -->  <aop:advisor advice-ref=*"exceptionAdvice"* pointcut-ref=*"pct"*/>  </aop:config>  </bean> |

### 常见的参数配置

## aspactJ开发

步骤：

1.目标类：接口 + 实现

2.切面类：编写多个通知，采用aspectj 通知名称任意（方法名任意）

3.aop编程，将通知应用到目标类

4.测试

### aspectJ的通知类型

 aop联盟定义通知类型，具有特性接口，必须实现，从而确定方法名称。

 aspectj 通知类型，只定义类型名称。已经方法格式。

 个数：6种，知道5种，掌握1中。

before:前置通知(应用：各种校验)

在方法执行前执行，如果通知抛出异常，阻止方法运行

afterReturning:后置通知(应用：常规数据处理)

方法正常返回后执行，如果方法中抛出异常，通知无法执行

必须在方法执行后才执行，所以可以获得方法的返回值。

around:环绕通知(应用：十分强大，可以做任何事情)

方法执行前后分别执行，可以阻止方法的执行

必须手动执行目标方法

afterThrowing:抛出异常通知(应用：包装异常信息)

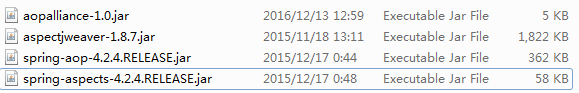
方法抛出异常后执行，如果方法没有抛出异常，无法执行

after:最终通知(应用：清理现场)

方法执行完毕后执行，无论方法中是否出现异常

|  |
| --- |
| 环绕  try{  //前置：before  //手动执行目标方法  //后置：afterRetruning  } catch(){  //抛出异常 afterThrowing  } finally{  //最终 after  } |

### 导入aspectjia包



### 定义目标类

|  |
| --- |
| **public** **class** Person{ //目标类  // PointCut 切入点  **public** **void** eat(){  System.*out*.println("吃饭");  }  } |

### 定义切面类

|  |
| --- |
| **public** **class** MyAspect {  //前置通知  **public** **void** beforeAdvice(){  System.*out*.println("前置通知:饭前刷牙");  }  //后置通知:reurnVal表示返回值  **public** **void** afterAdvice(Object reurnVal){  System.*out*.println("后置通知:饭后广场舞，目标方法返回值："+reurnVal);  }  //环绕通知  **public** Object aroundAdvice(ProceedingJoinPoint pj){  Object result=**null**;  **try** {  Long start=System.*currentTimeMillis*();    System.*out*.println("目标方法调用前的时间："+start);  //调用目标对像的方法  result = pj.proceed();  //获取目标类的名称  String name= pj.getTarget().getClass().getName();  Long end=System.*currentTimeMillis*();    System.*out*.println("目标方法"+name+"执行完后的时间："+end+",调用方法耗时："+(end-start)+"毫秒");    } **catch** (Throwable e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **return** result;  }  //异常通知  **public** **void** execptionAdvice(Throwable t){  System.*out*.println("产生了异常，异常消息："+t.getMessage());  }  } |

### xml配置

|  |
| --- |
| <!-- 目标类的对象 -->  <bean id=*"person"* class=*"cn.sxt.domain.Person"*></bean>  <!-- 前置通知的对象 -->  <bean id=*"beforeadvice"* class=*"cn.sxt.advice.MyBeforeAdvice"*></bean>  <!-- 后置通知的对象 -->  <bean id=*"afteradvice"* class=*"cn.sxt.advice.MyAfterAdvice"*></bean>  <!-- 环绕通知 -->  <bean id=*"arounddvice"* class=*"cn.sxt.advice.MyAroundAdvice"*></bean>  <!-- 异常通知 -->  <bean id=*"exceptionAdvice"* class=*"cn.sxt.advice.MyExceptionAdvice"*>  <!-- aop的配置 -->  <aop:config>  <!-- 切入点：要增强的方法 expression:切入点表达式 切入点表达式规则： execution(返回类型 包名.类名.方法名(形参)  ) -->  <aop:pointcut expression=*"execution(void cn.sxt.domain.Person.eat(..))"*  id=*"pct"* />  <!-- aop:advisor:完成织入，组合切入点和通知 advice-ref:引用Advice实例,通过id引用 pointcut-ref:引用切入点 -->  <!-- <aop:advisor advice-ref="beforeadvice" pointcut-ref="pct" />  <aop:advisor advice-ref="afteradvice" pointcut-ref="pct" /> -->  <!-- <aop:advisor advice-ref="arounddvice" pointcut-ref="pct" /> -->  <aop:advisor advice-ref=*"exceptionAdvice"* pointcut-ref=*"pct"*/>  </aop:config>  </bean> |

### 前置

格式：:before method="" pointcut="" pointcut-ref=""/>

method : 通知，及方法名

pointcut :切入点表达式，此表达式只能当前通知使用。

pointcut-ref ： 切入点引用，可以与其他通知共享切入点。

知方法格式：public void myBefore(JoinPoint joinPoint){

数1：org.aspectj.lang.JoinPoint 用于描述连接点（目标方法），获得目标方法名等

### 后置‘

目标方法后执行，获得返回值

格式：<aop:after-returning method="" pointcut-ref="" returning=""/>

returning 通知方法第二个参数的名称

通知方法格式：public void myAfterReturning(JoinPoint joinPoint,Object ret){

参数1：连接点描述

参数2：类型Object，参数名 returning="ret" 配置的

### ’环绕

格式：<aop:around method="" pointcut-ref=""/>

通知方法格式：public Object myAround(ProceedingJoinPoint joinPoint) throws Throwable{

返回值类型：Object

方法名：任意

参数：org.aspectj.lang.ProceedingJoinPoint

抛出异常

执行目标方法：Object obj = joinPoint.proceed();

### 异常通知

格式：<aop:after-throwing method="" pointcut-ref="" throwing=""/>

throwing ：通知方法的第二个参数名称

通知方法格式：public void myAfterThrowing(JoinPoint joinPoint,Throwable e){

参数1：连接点描述对象

参数2：获得异常信息，类型Throwable ，参数名由throwing="e" 配置

## aspectJ的注解

### 替换bean(目标类和切面类)

xml中

|  |
| --- |
| <!-- 目标类的对象 -->  <bean id=*"person"* class=*"cn.sxt.domain.Person"*></bean>  <!-- 前置通知的对象 -->  <bean id=*"beforeadvice"* class=*"cn.sxt.advice.MyBeforeAdvice"*></bean> |

替换为

|  |
| --- |
| @Component  public class Person{} |

### 扫描注解（注意配置扫描）

|  |
| --- |
| <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd*  *"*>  1、扫描切面类、和serviceimpl  <context:component-scan base-package=*"com.sx.\*"*></context:component-scan>  2、开启基于注解的aop功能  <aop:aspectj-autoproxy></aop:aspectj-autoproxy> |

### 替换oap

1、替换切面类

|  |
| --- |
| @Component //表明类要被spring去实例化  @Aspect //表明类是切面类，代替配置文件中的aop:aspect元素  public class MyAspect {  } |

2、前置通知

|  |
| --- |
| @Before("execution(\* cn.sxt.domain.\*.\*(..))")  public void beforeAdvice(){  System.out.println("前置通知:饭前刷牙");  } |

2、后置通知

|  |
| --- |
| @AfterReturning(pointcut= "execution(\* cn.sxt.domain.\*.\*(..))",returning="returnvalue")  public void afterAdvice(Object returnvalue){  System.out.println("后置通知:饭后广场舞,目标方法返回值："+returnvalue);  } |

2、替换前置

|  |
| --- |
| @Around("execution(\* cn.sxt.domain.\*.\*(..))") |

2、异常通知

|  |
| --- |
| @AfterThrowing(pointcut= "execution(\* cn.sxt.domain.\*.\*(..))",throwing="t") |

## 切入点表达式

### execution()用于描述方法 【掌握】可以锁定到方法上面去。

语法：execution(修饰符 返回值 包.类.方法名(参数) throws异常)

修饰符，一般省略

public 公共方法

\* 任意

返回值，不能省略

void 返回没有值

String 返回值字符串

\* 任意

包，[省略]

com.itheima.crm 固定包

com.itheima.crm.\*.service crm包下面子包任意 （例如：com.itheima.crm.staff.service）

com.itheima.crm.. crm包下面的所有子包（含自己）

com.itheima.crm.\*.service.. crm包下面任意子包，固定目录service，service目录任意包

类，[省略]

UserServiceImpl 指定类

\*Impl 以Impl结尾

User\* 以User开头

\* 任意

方法名，不能省略

addUser 固定方法

add\* 以add开头

\*Do 以Do结尾

\* 任意

(参数)

() 无参

(int) 一个整型

(int ,int) 两个

(..) 参数任意

throws ,可省略，一般不写。

综合1

execution(\* com.itheima.crm.\*.service..\*.\*(..))

返回值任意，crm的子包任意，模块里进行分层，分为web层，dao层，service层等。service文件夹下存放对应的接口和实现类。实现类的名字是任意，方法也是任意。参数任意。

综合2

<aop:pointcut expression="execution(\* com.itheima.\*WithCommit.\*(..)) ||

execution(\* com.itheima.\*Service.\*(..))" id="myPointCut"/>

使用||可以实现或操作，匹配两个操作。

2.within:匹配包或子包中的方法(了解)

within(com.itheima.aop..\*)

3.this:匹配实现接口的代理对象中的方法(了解)

this(com.itheima.aop.user.UserDAO)

4.target:匹配实现接口的目标对象中的方法(了解)

target(com.itheima.aop.user.UserDAO)

5.args:匹配参数格式符合标准的方法(了解)

args(int,int)

6.bean(id) 对指定的bean所有的方法(了解)

bean('userServiceId')

## 总结

1、从ioc容器中拿到目标对象；注意：如果想要用类型，一定用他的接口类型，不要用它本类

细节一：com.atguigu.impl.MyMathcalculator@6c64cb25

class com.sun.proxy.$Proxy12

AOP的底层就是动态代理，容器中保存的组件是他的代理对象；$Proxy12。当然不是本类的类型。

所以不能使用本类去去getBean,只能使用接口去获取代理对象或者使用实现类小写去获取代理对象，但是不能强转为实现类。

如果有接口转为接口类型，没有接口转为实现类类型。

2、切面表达式正确写法

固定格式：execution（访问权限符 返回值类型 方法全类名（参数表））

通配符：

\*，

1）匹配一个或者多个字符

execution(public int com.sxt.impl.userService\*impl.\*.\*(int,int)

2)匹配任意一个参数

execution(public int com.sxt.impl.userService\*impl.\*.\*(int,\*)

第一个参数为int，第二个为任意，有且只有两个参数。

. .

1)匹配任意多参数

execution(public int com.sxt.impl.userService\*impl.\*.\*(..)

2)匹配任意路径

execution(public int com.sxt.\*.userService\*impl.\*.\*(..)该\*只能匹配一层路径

execution(public int com.sxt..userService\*impl.\*.\*(..)该..匹配任意路径

### 方法的执行顺序

|  |
| --- |
| try{  //前置：before  //手动执行目标方法  //后置：afterRetruning  } catch(){  //抛出异常 afterThrowing  } finally{  //最终 after  } |

正常执行顺序

@Before 前置通知

@After 后置通知（最终结束）

@AfterRetruning (后置通知)

异常执行顺序

@Before 前置通知

@After 后置通知（最终结束）

@AfterThrowing (方法异常)

### 通知方法信息获取

我们可以在通知方法运行的时候，拿到目标方法的详细信息；

1)只需要为通知方法的参数列表上写一个参数：JoinPoint joinPoint:封装了当前目标方法的详细信息。

|  |
| --- |
| @Around("execution(\* com.sx.service.impl.\*.\*(..))")  **public** **void** aroundAdvice(ProceedingJoinPoint joinPoint) {  System.***out***.println("环绕通知");  **try** {  //1、获取方法的参数  Object[] args = joinPoint.getArgs();  **for** (Object object : args) {  System.***out***.println(object);  }  //2、获取方法的签名  Signature signature = joinPoint.getSignature();  //方法名  String name = signature.getName();  //方法的修饰符  **int** modifiers = signature.getModifiers();  System.***out***.println(name+modifiers);  joinPoint.proceed();  } **catch** (Throwable e) {  e.printStackTrace();  }  System.***out***.println("后置通知");  } |

### 调用方法的返回值

唯一要求的就是方法的参数列表一定不能乱写?

通知方法是Spring利用反射调用的，每次方法调用得确定这个方法的参数表的值；参数表上的每一个参数，Spring都得知道是什么?

1）JoinPoint

JoinPoint joinPoint:封装了当前目标方法的详细信息

2）、告诉Spring哪个参数是用来接收异常

throwing="exception":告诉Spring哪个参数是用来接收异常

3）、Exception exception:指定通知方法可以接收哪些异常

### 切入点表达式的重用

抽取可重用的切入点表达式；

1、随便声明一个没有实现的返回void的空方法

2、给方法上标注@Pointcut注解

|  |
| --- |
| @Pointcut("execution(\* com.sx.service.impl.\*.\*(..))")  **public** **void** pointcut() {};    //引用上边的切入点表达式  @Before("pointcut()")  **public** **void** beforeAdvice( ) {  System.***out***.println("前置通知");  } |

### 环绕通知

环绕通知是由于普通通知执行。

执行顺序

|  |
| --- |
| [普通前置]  **try** {  [环绕前置]  [环绕执行:目标方法执行]  [环绕返回]  } **catch** (Exception e) {  [环绕出现异常]  }**finally** {  [环绕后置]  }  [普通后置]  [普通方法返回/方法异常] |

(环绕前置-—-普通前置)----目标方法执行-—-环绕正常返回/出现异常-—-环绕后置-—-普通后置---普通返回或者异常

1、(环绕前置-—-普通前置顺序无所谓，但是后边的的通知顺序严格

2、环绕通知出现异常，会把这个异常吃掉，所以一定要将环绕的异常排除，后边的异常会接收掉。

|  |
| --- |
| 环绕通知：就是下列四个通知一起构成。  try{  //前置：before  //手动执行目标方法  //后置：afterRetruning  } catch(){  //抛出异常 afterThrowing  } finally{  //最终 after  } |

环绕通知

|  |
| --- |
| @Around("execution(\* com.sx.service.impl.\*.\*(..))")  **public** Object aroundAdvice(ProceedingJoinPoint joinPoint) {    Object proceed = **null**;  **try** {  System.***out***.println("环绕前置");  Object[] args2 = joinPoint.getArgs();  proceed = joinPoint.proceed(args2); //方法返回值  System.***out***.println("环绕返回通知");  } **catch** (Throwable e) {  e.printStackTrace();  System.***out***.println("环绕异常");  }  System.***out***.println("环绕后置通知");  **return** proceed; //这里的返回值，一定是方法的返回值。  } |

# jdbcTemplate

## jdbcTeamlate概述

Spring为各种支持的持久化技术，都提供了简单操作的模板和回调

|  |  |
| --- | --- |
| ORM持久化技术 | 模板类 |
| JDBC | org.springframework.jdbc.core.JdbcTemplate |
| Hibernate5.0 | org.springframework.orm.hibernate5.HibernateTemplate |
| IBatis(MyBatis) | org.springframework.orm.ibatis.SqlMapClientTemplate |
| JPA | org.springfrmaework.orm.jpa.JpaTemplate |

其实Spring的JDBCTemplate有点像DBUtils，但是有时候还没有DBUitls好用。这里来学习一下使用Spring的JDBCTemplate来玩一下CRUD。

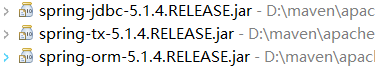
## TempLate的Api

|  |
| --- |
| int update():执行增，删，改  List<T> query(String sql, RowMapper<T> rowMapper)：针对多行多列的查询， 返回List集合，不带输入参数(不带?)  List<T> query(String sql, RowMapper<T> rowMapper, Object... args)：针对多行多列的查询， 返回List集合，带输入参数  T queryForObject(String sql, RowMapper<T> rowMapper, Object... args) :针对单行多列的查询，返回一个实体对象  T queryForObject(String sql, Class<T> requiredType, Object... args):针对单行单列的查询，返回单个值（String/Integer/Long等），Class<T>是指定返回的java类型  List<T> queryForList(String sql, Class<T> elementType, Object... args)：针对多行单列的查询，参数2是集合中元素的类型（String/Integer/Long等)，返回一个集合 |

## 入门程序

### 导入spring的jdbc需要的jar包

jdbc包、tx事务包，orm包



jdbc驱动包

image

### javabean

|  |
| --- |
| **public** **class** User {  **private** Integer id;  **private** String name;  **private** String sex;  **private** String address;  …  } |

### jdbcTeaplate入门

|  |
| --- |
| **public** **class** SpringJdbcTemplateTest {  @Test  **public** **void** test1() {  // 1.创建数据源(提供连接数据库参数)  DriverManagerDataSource dataSource = **new** DriverManagerDataSource();  dataSource.setDriverClassName("com.mysql.jdbc.Driver");  dataSource.setUrl("jdbc:mysql://127.0.0.1:3306/spring\_db");  dataSource.setUsername("root");  dataSource.setPassword("root");  // 2.创建JdbcTemplate，它要绑定数据源  JdbcTemplate jdbcTemplate = **new** JdbcTemplate();  jdbcTemplate.setDataSource(dataSource);  // 3.调用JdbcTemplate的api去完成crud  // 获取表的总记录数  Integer num = jdbcTemplate.queryForObject(  "select count(\*) from t\_user", Integer.**class**);  System.*out*.println("总记录数：" + num);  String sql="select \* from t\_user";  //定义行映射器 Rowmapper,如果字段与属性名完全一致，可以用BeanPropertyRowMapper(把字段的值封装到Bean对象的同名属性上)  RowMapper<User> rowMapper=**new** BeanPropertyRowMapper<User>(User.**class**);  List<User> list = jdbcTemplate.query(sql, rowMapper);  System.*out*.println(list.size());  }  } |

其实这个例子本身没有什么的，只是演示了一下，其实在学Spring之后，感觉应该形成一种习惯，在new对象的时候我要想到IOC，在使用Set方法的时候，我要想到DI，再去要方便面（哦，不，是切面），我们应该想到用AOP的。这里可以在Spring中配置如下的引用链：

1. 我要有DataSource，DataSource的属性可以通过注入数据库的一些配置属性添加

2. 我要有JdbcTemplate，而Template依赖与DataSource，我要以ref的方式为我的JdbcTemplate注入引用

3. 有了JdbcTemplate之后，我要有Dao，此时我应该在Dao添加一个JdbcTemplate的成员，然后以ref的方式将JdbcTemplate引入到Dao中

4. 我在Action或者是Servlet中都会调用的是Serivce，所以，我在Serivce中要添加一个Dao作为成员，然后由ref在注入Dao到Service中

DataSource --> JdbcTemplate --> Dao --> Service --> Action/Servlet

"-->"表示将左边的对象注入到右边的对象当中

### 配置appicationContext

约束

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd*  *"*>  <!-- 配置数据源 -->  <bean id=*"myDataSource"*  class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>  <property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*></property>  <property name=*"url"* value=*"jdbc:mysql://127.0.0.1:3306/spring\_db"*></property>  <property name=*"username"* value=*"root"*></property>  <property name=*"password"* value=*"root"*></property>  </bean>  <!-- 配置JdbcTemplate对象 -->  <bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <property name=*"dataSource"* ref=*"myDataSource"* ></property>  </bean>  <!-- 配置Dao -->  <bean id=*"userDao"* class=*"cn.sxt.dao.impl.UserDaoImpl"*>  <property name=*"jdbcTemplate"* ref=*"jdbcTemplate"*></property>  </bean>  <!-- 配置Service -->  <bean id=*"userService"* class=*"cn.sxt.service.impl.UserServiceImpl"*>  <property name=*"userDao"* ref=*"userDao"*></property>    </bean>  <!-- 配置Controller -->  <bean id=*"userController"* class=*"cn.sxt.controller.UserController"* autowire=*"byType"*>  <!-- <property name="userService" ref="userService"></property> -->  </bean>  </beans> |

### 接口方法

|  |
| --- |
| **public** **interface** UserDao {  // 插入  **int** insert(User user);  // 修改  **int** update(User user);  // 删除  **int** delete(Integer id);  // 全查询  List<User> queryAll();  // 查询单个记录  User queryById(Integer id);  // 分页查询，获取当前页数据集合  List<User> queryForPage(**int** currentPage, **int** pageSize);  // 分页查询，获取总记录数  Integer queryCount();  } |

### 实现类（字段名和属性名是相同的）

|  |
| --- |
| **public** **class** UserDaoImpl **implements** UserDao {  //这里使用到JdbcTemplate模板对象，由spring依赖注入。并提供setter方法。  **private** JdbcTemplate jdbcTemplate;  **public** JdbcTemplate getJdbcTemplate() {  **return** jdbcTemplate;  }  **public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {  **this**.jdbcTemplate = jdbcTemplate;  }  @Override  **public** **int** insert(User user) {  // **TODO** Auto-generated method stub  String sql = "insert into t\_user(name,sex,address) values(?,?,?)";  Object[] params = { user.getName(), user.getSex(), user.getAddress() };  **int** i = jdbcTemplate.update(sql, params);  **return** i;  }  @Override  **public** **int** update(User user) {  String sql = "update t\_user set name=?,sex=?,address=? where id=?";  Object[] params = { user.getName(), user.getSex(), user.getAddress(),  user.getId() };  **int** i = jdbcTemplate.update(sql, params);  **return** i;  }  @Override  **public** **int** delete(Integer id) {  String sql = "delete from t\_user where id=?";  Object[] params = { id };  **int** i = jdbcTemplate.update(sql, params);  **return** i;  }  @Override  **public** List<User> queryAll() {  // **TODO** Auto-generated method stub  String sql="select \* from t\_user";  //定义行映射器 Rowmapper,如果字段与属性名完全一致，可以用BeanPropertyRowMapper(把字段的值封装到Bean对象的同名属性上)  RowMapper<User> rowMapper=**new** BeanPropertyRowMapper<User>(User.**class**);  List<User> list = jdbcTemplate.query(sql, rowMapper);  **return** list;  }  @Override  **public** User queryById(Integer id) {  // **TODO** Auto-generated method stub  String sql="select \* from t\_user where id=?";  RowMapper<User> rowMapper=**new** BeanPropertyRowMapper<User>(User.**class**);  Object[] params={id};  User user = jdbcTemplate.queryForObject(sql, rowMapper, params);  **return** user;  }  @Override  **public** List<User> queryForPage(**int** currentPage, **int** pageSize) {  String sql="select \* from t\_user limit ?,?";  Object[] params={(currentPage-1)\*pageSize,pageSize};  RowMapper<User> rowMapper=**new** BeanPropertyRowMapper<User>(User.**class**);  List<User> list = jdbcTemplate.query(sql, rowMapper,params);    **return** list;  }  @Override  **public** Integer queryCount() {  // **TODO** Auto-generated method stub  String sql="select count(\*) from t\_user";  Integer num = jdbcTemplate.queryForObject(sql, Integer.**class**);    **return** num;  }  } |

### service

|  |
| --- |
| **public** **interface** UserService {  // 插入  **boolean** addUser(User user);  // 修改  **boolean** updateUser(User user);  // 删除  **boolean** deleteUser(Integer id);  // 获取所有用户  List<User> queryUsers();  // 查询单个记录  User queryUserById(Integer id);  // 分页查询，获取当前页数据集合  List<User> queryForPage(**int** currentPage, **int** pageSize);  // 分页查询，获取总记录数  Integer queryCount();  } |

### serviceimpl

|  |
| --- |
| **public** **class** UserServiceImpl **implements** UserService {  //添加UserDao ，配置中使用依赖注入  **private** UserDao userDao;    **public** UserDao getUserDao() {  **return** userDao;  }  **public** **void** setUserDao(UserDao userDao) {  **this**.userDao = userDao;  }  ……….  } |

### controller

|  |
| --- |
| **public** **class** UserController {  //设置userService，由spring注入。  **private** UserService userService;  **public** UserService getUserService() {  **return** userService;  }  **public** **void** setUserService(UserService userService) {  **this**.userService = userService;  }  ….  } |

### 测试

|  |
| --- |
| **public** **class** SpringTest {  **public** **static** **void** main(String[] args) {  //获取ApplictionContext  ApplicationContext ctx=**new** ClassPathXmlApplicationContext("classpath:applicationContext.xml");  //获取web层的控制器  UserController bean =(UserController) ctx.getBean("userController");  } |

### 字段名与属性名不相同

如果字段名和属性名不一致，那么不能直接用BeanPropertyRowMapper来映射行，需要自己去重写RowMapper里面的mapRow方法，在方法里面去指定字段与属性的映射规则。

|  |
| --- |
| // 测试对部门表的查询，字段名和属性名不相同  @Test  **public** **void** test2() {  // 从ioc容器中拿jdbcTemplate对象  ApplicationContext ctx = **new** ClassPathXmlApplicationContext(  "classpath:applicationContext.xml");  JdbcTemplate jdbcTemplate = (JdbcTemplate) ctx.getBean("jdbcTemplate");  String sql = "select \* from t\_dept";  RowMapper<Dept> rowMapper = **new** RowMapper<Dept>() {  @Override//内部类  **public** Dept mapRow(ResultSet rs, **int** arg1) **throws** SQLException {  Dept dept = **new** Dept();  **int** did = rs.getInt("did");  String dname = rs.getString("d\_name");  dept.setDid(did);  dept.setDname(dname);  **return** dept;  }  };  List<Dept> query = jdbcTemplate.query(sql, rowMapper);    **for** (Dept dept : query) {  System.*out*.println(dept.getDid() + "..." + dept.getDname());  }  } |

## 注解方式

1、配置xml

|  |
| --- |
| <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd*  *"*>  <!-- 配置数据源 -->  <bean id=*"myDataSource"*  class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>  <property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*></property>  <property name=*"url"* value=*"jdbc:mysql://127.0.0.1:3306/spring\_db"*></property>  <property name=*"username"* value=*"root"*></property>  <property name=*"password"* value=*"root"*></property>  </bean>  <!-- 配置JdbcTemplate对象 -->  <bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <property name=*"dataSource"* ref=*"myDataSource"* ></property>  </bean>    <!-- 组件扫描 -->  <context:component-scan base-package=*"cn.sxt.dao.impl,cn.sxt.service.impl,cn.sxt.controller"*></context:component-scan>  </beans> |

### 注入

其他的使用注解依次注入

Dao:@repository

Service:@Service

Controller:@ Controller

属性：@Autowired

## 四种数据源

### spring自带的数据源

|  |
| --- |
| <bean id=*"myDataSource"*  class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>  <property name=*"driverClassName"* value=*"${driver}"*></property>  <property name=*"url"* value=*"${url}"*></property>  <property name=*"username"* value=*"${username}"*></property>  <property name=*"password"* value=*"${password}"*></property>  </bean> |

# Spring的定时任务

## 定时任务概述

### Quartz介绍

在企业应用中，我们经常会碰到时间任务调度的需求，比如每天凌晨生成前天报表，每小时生成一次汇总数据等等。Quartz是出了名的任务调度框架,它可以与J2SE和J2EE应用程序相结合，功能灰常强大，轻轻松松就能与Spring集成，使用方便。

### Quartz中的概念

　主要有三个核心概念：调度器、任务和触发器。三者关系简单来说就是，调度器负责调度各个任务，到了某个时刻或者过了一定时间，触发器触动了，特定任务便启动执行。概念相对应的类和接口有：

　1）JobDetail：望文生义就是描述任务的相关情况；

2)Trigger：描述出发Job执行的时间触发规则。有SimpleTrigger和CronTrigger两个子类代表两种方式，一种是每隔多少分钟小时执行，则用SimpleTrigger；另一种是日历相关的重复时间间隔，如每天凌晨，每周星期一运行的话，通过Cron表达式便可定义出复杂的调度方案。

　3)Scheduler：代表一个Quartz的独立运行容器，Trigger和JobDetail要注册到Scheduler中才会生效，也就是让调度器知道有哪些触发器和任务，才能进行按规则进行调度任务。

## xml配置JobDteail通过JobDetailBean实现

### 任务类

|  |
| --- |
| **public** **class** MyJob1 {  //任务方法  **public** **void** printTask(){  Date date = **new** Date();  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  System.*out*.println("定时任务..当前时间：" + sdf.format(date));  }  } |

### application

|  |
| --- |
| <!-- 配置任务 -->  <bean id=*"job1"* class=*"cn.sxt.job.MyJob1"*></bean>  <!-- 任务详情，包装任务的具体情况 -->  <bean id=*"myJobDetail"* class=*"org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean"*>  <!-- 配置目标任务 -->  <property name=*"targetObject"* ref=*"job1"* ></property>  <!-- 配置任务中执行的方法 -->  <property name=*"targetMethod"* value=*"printTask"*></property>  </bean>  <!-- 触发器 -->  <bean id=*"triggerbean"* class=*"org.springframework.scheduling.quartz.CronTriggerFactoryBean"*>  <!-- 任务详情 -->  <property name=*"jobDetail"* ref=*"myJobDetail"*></property>  <!-- 用cronExpression描述任务执行的时间规则 -->  <property name=*"cronExpression"* value=*"0/5 \* \* \* \* ? "*></property>  </bean>  <!-- 调度器 -->  <bean id=*"schedulerBean"* class=*"org.springframework.scheduling.quartz.SchedulerFactoryBean"*>  <property name=*"triggers"*>  <list>  <ref bean=*"triggerbean"*/>  </list>  </property>  </bean> |

### 测试

|  |
| --- |
| **public** **class** QuartzTest {  **public** **static** **void** main(String[] args) {  //获取ApplictionContext,IOC容器被加载后会自动执行定时任务  ApplicationContext ctx=**new** ClassPathXmlApplicationContext("classpath:applicationContext.xml");  }  } |

## xml配置JobDteail通过MethodInvokeJobDetailFactoryBean实现(推荐)

### 任务

|  |
| --- |
| **public** **class** MyJob2 **extends** QuartzJobBean{  //任务方法  @Override  **protected** **void** executeInternal(JobExecutionContext arg0)  **throws** JobExecutionException {  Date date = **new** Date();  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  System.*out*.println("定时任务2..当前时间：" + sdf.format(date));  }  } |

### 配置applicationContext.xml

|  |
| --- |
| <!-- 任务详情，包装任务的具体情况 -->  <bean id=*"myJobDetail"* class=*"org.springframework.scheduling.quartz.JobDetailFactoryBean"*>  <!-- 注入任务类,value:Job的全限定名 -->  <property name=*"jobClass"* value=*"cn.sxt.job.MyJob2"*></property>  <!-- 是否重复执行 -->  <property name=*"durability"* value=*"true"*></property>  </bean>  <!-- 触发器 -->  <bean id=*"triggerbean"* class=*"org.springframework.scheduling.quartz.CronTriggerFactoryBean"*>  <!-- 任务详情 -->  <property name=*"jobDetail"* ref=*"myJobDetail"*></property>  <!-- 用cronExpression描述任务执行的时间规则 -->  <property name=*"cronExpression"* value=*"20 36 14 1 \* ? "*></property>  </bean>  <!-- 调度器 -->  <bean id=*"schedulerBean"* class=*"org.springframework.scheduling.quartz.SchedulerFactoryBean"*>  <property name=*"triggers"*>  <list>  <ref bean=*"triggerbean"*/>  </list>  </property>  </bean> |

### 测试

|  |
| --- |
| **public** **class** QuartzTest {  **public** **static** **void** main(String[] args) {  //获取ApplictionContext,IOC容器被加载后会自动执行定时任务  ApplicationContext ctx=**new** ClassPathXmlApplicationContext("classpath:applicationContext.xml");  }  } |

## 注解方式

### 任务类

|  |
| --- |
| @Coponent //添加注解  **public** **class** MyJob3 {  // 任务方法  @Scheduled(cron = "0/5 \* \* \* \* ? ") //添加定时任务注解  **protected** **void** doTask(){  Date date = **new** Date();  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  System.*out*.println("定时任务3..当前时间：" + sdf.format(date));  }  } |

### applicationContext.xml配置

|  |
| --- |
| <!-- 开启组件扫描 -->  <context:component-scan base-package=*"cn.sxt.job"*></context:component-scan>  <!-- 开启支持定时任务的注解 -->  <task:annotation-driven /> |

# spring事务管理

如果实现了事务功能和spring的整合，他的对象是实现了service接口的代理类对象。该对象右ioc生成，

## spring事务

### 事务的基本原理

Spring事务 的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。对于纯JDBC操作数据库，想要用到事务，可以按照以下步骤进行：

获取连接 Connection con = DriverManager.getConnection()

开启事务con.setAutoCommit(true/false);

执行CRUD

提交事务/回滚事务 con.commit() / con.rollback();

关闭连接 conn.close();

使用Spring的事务管理功能后，我们可以不再写步骤 2 和 4 的代码，而是由Spirng 自动完成。 那么Spring是如何在我们书写的 CRUD 之前和之后开启事务和关闭事务的呢？解决这个问题，也就可以从整体上理解Spring的事务管理实现原理了。下面简单地介绍下，注解方式为例子

配置文件开启注解驱动，在相关的类和方法上通过注解@Transactional标识。

spring 在启动的时候会去解析生成相关的bean，这时候会查看拥有相关注解的类和方法，并且为这些类和方法生成代理，并根据@Transaction的相关参数进行相关配置注入，这样就在代理中为我们把相关的事务处理掉了（开启正常提交事务，异常回滚事务）。

真正的数据库层的事务提交和回滚是通过binlog或者redo log实现的。

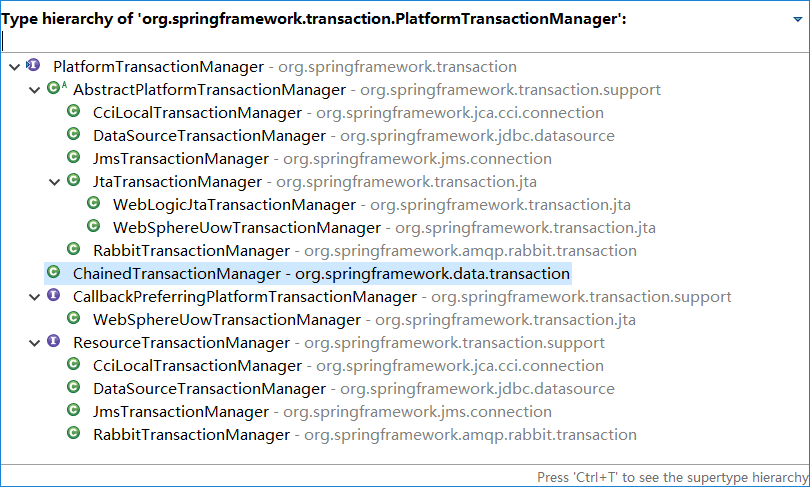
### 声明式事务：

以前通过复杂的编程来编写一个事务，替换为只需要告诉Spring哪个方法是事务方法即可；Spring自动进行事务控制；

### 实现原理

事务管理代码的固定模式作为一种横切关注点，可以通过AOP方法模块化，进而借助Spring AOP框架实现声明式事务管理。

事务切面就是事务管理器。

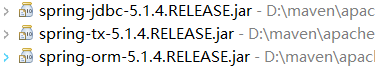


不同的框架可以使用不同的事务管理器。目前使用的是dataSourceTransactionManager

## helloworld

### 导入spring的jdbc需要的jar包

jdbc包、tx事务包，orm包



jdbc驱动包

image

### 配置事务管理器

|  |
| --- |
| <!--事务管理将数据控制 -->  <bean id =*"transactionManager"*class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>  <!--开启基于注解的事务控制模式，tx名称空间 -->  <tx:annotation-driven transaction-manager=*"transactionManager"*/>  <!--给事务方法加注解 --> |

## 事务注解

### 异常分类：

运行时异常（非检查异常）：可以不用处理；默认都回滚；

编译时异常（检查异常）：要么try-catch，要么在方法上声明throws

事务的回滚：默认发生运行时异常都回滚，发生编译时异常不会回滚；

### @Transactional

事务细节：

isolation-Isolation:事务的隔离级别

propagation-Propagation:事务的传播行为

noRollbackFor-Class[]:那些异常事务可以不会滚（让那些默认回滚的异常可以不会滚）

noRollbackForClassName-String[]:

rollbackFor-Class[]:那些异常事务需要回滚

rollbackForClassName-String[]:

readOnly-boolean: 设置事务为只读事务，

进行事务优化，readOnly=true，可以加快查询速度，不用管事务的那些操作了。

如果在方法中增删该等操作，使用这个属性会出错。

timeout-int（秒为单位）:超时，事务超过特定执行时长，后自动终止并回滚

## 事务的传播特性

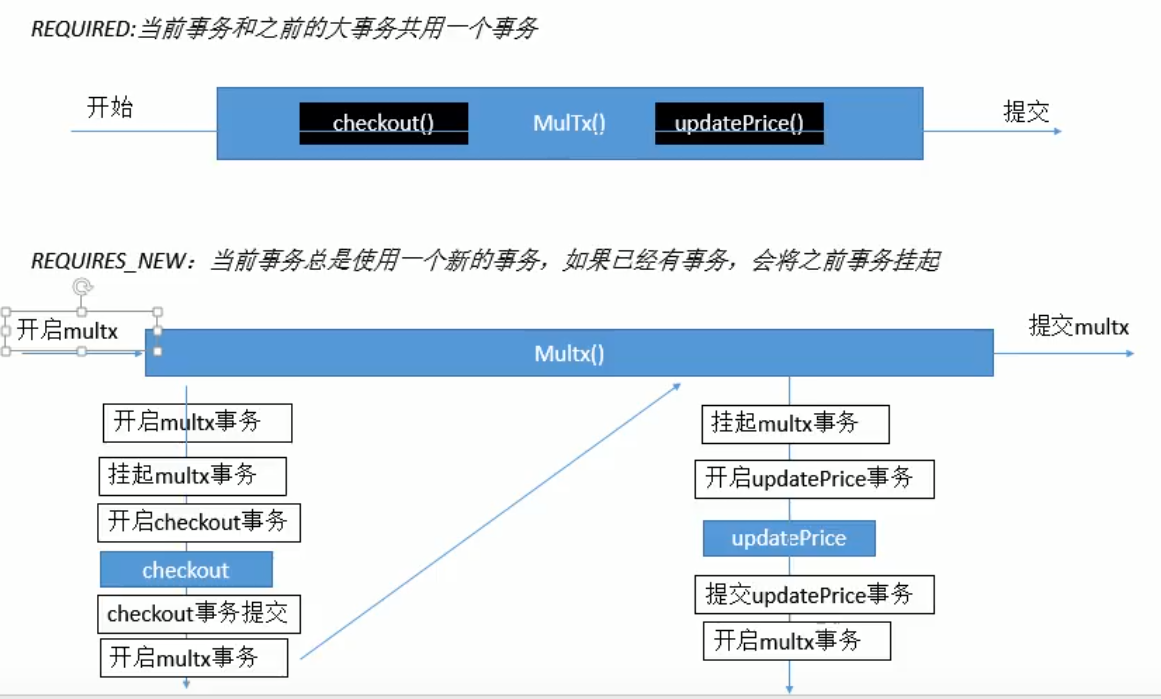
传播行为(事务的传播+事务的行为) ;如果有多个事务进行嵌套运行,子事务是否要和大事务共用一个事务。

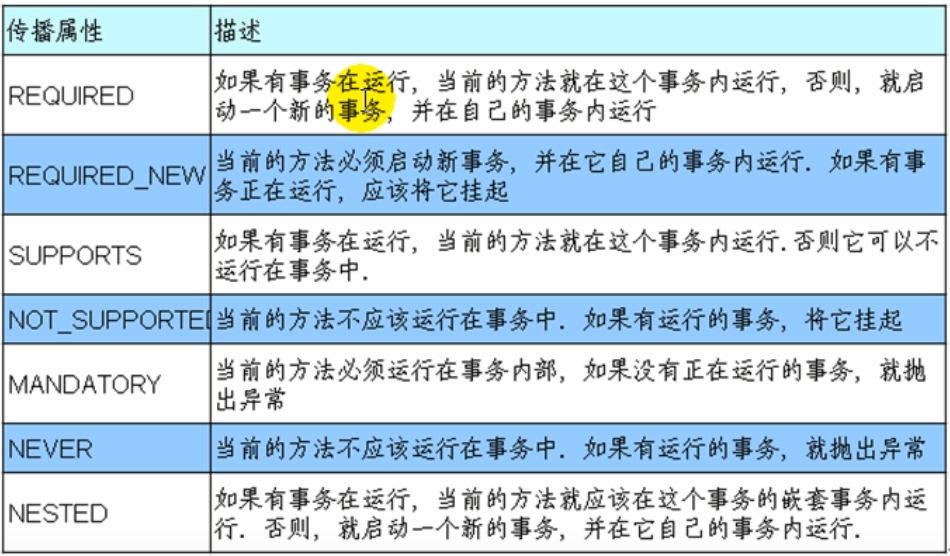
所谓事务传播特性就是多个事务方法相互调用时，事务如何在这些方法间传播。

如果是requeired，子事务的属性时继承于大事务的。

REQUIRED:将之前事务用的connection传递给这个方法使用;

REQUIRES NEW:这个方法直接使用新的connection;





# spring底层原理实现

# 附录

## 分页插件pageHelper

### 导入jar包



### 在mybatis的配置文件中配置分页插件

|  |
| --- |
| <plugins>  <!-- 配置分页插件，定义拦截sql的拦截器 -->  <plugin interceptor=*"com.github.pagehelper.PageInterceptor"*>  </plugin>  </plugins> |

### 在UserMapper接口加入针对分页查询的方法，只要查符合条件的集合

|  |
| --- |
| **public** **interface** UserMapper {  //根据条件做模糊查询  List<User> queryByCondition(User user);  } |

### UserMapper.xml文件中，对应的statement

|  |
| --- |
| <select id=*"queryByCondition"* parameterType=*"User"* resultType=*"User"*>  select \* from t\_user  <where>  <if test=*"name!=null"*>  and name like '%${name}%'  </if>  <if test=*"sex!=null"*>  and sex=#{sex}  </if>  </where>  </select> |

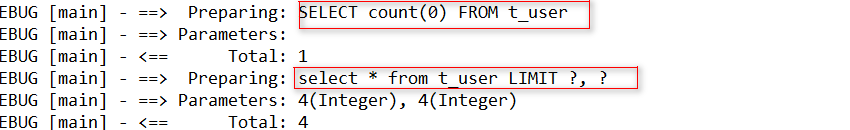
### 在UserService接口中加入方法，调用UserMapper,在调用UserMapper的方法前一定要设置分页参数，否则分页插件不起作用。

|  |
| --- |
| **public** List<User> getUserByCondition(User user, PageBean<User> pageBean) {  // 设置分页插件的参数，目的是启动分页插件，当sql语句执行时会被拦截  // 参数1：分页页号 参数2：每页条数  **int** pageNum = pageBean.getCurrentPage();  **int** pageSize = pageBean.getPageSize();  Page<User> page = PageHelper.startPage(pageNum, pageSize);  List<User> list = userMapper.queryByCondition(user);  System.out.println("总记录数：" + page.getTotal() + ",每页条数："  + page.getPageSize());  **return** list;  } |

### 在UserController中调用Service的方法

|  |
| --- |
| **public** **void** getAllUsers() {  User user = **new** User();  //user.setName("王");  PageBean<User> pageBean = **new** PageBean<User>();  // 设置页号  pageBean.setCurrentPage(2);  // 设置每页条数  pageBean.setPageSize(4);  List<User> all = userService.getUserByCondition(user, pageBean);  **for** (User user2 : all) {  System.out.println(user2);  }  } |

### 结果



### pagehelper返回值问题